fabrics could be utilised within the printer roll 42 for printing images on fabric, although care must be taken that only fabrics having a suitable stiffness or suitable backing material are utilised.

When the print media is plastic, it can be coated with a layer which fixes and absorbs the ink. Further, several types of print media may be used, for example, opaque white matte, opaque white gloss, transparent film, frosted transparent film, lenticular array film for stereoscopic 3D prints, metallised film, film with the embossed optical variable devices such as gratings or holograms, media which is pre-printed on the reverse side, and media which includes a magnetic recording layer. When utilising a metallic foil, the metallic foil can have a polymer base, coated with a thin (several micron) evaporated layer of aluminum or other metal and then coated with a clear protective layer adapted to receive the ink via the ink printer mechanism.

In use the print roll 42 is obviously designed to be inserted inside a camera device so as to provide ink and paper for the printing of images on demand. The ink output ports 635 - 637 meet with corresponding ports within the camera device and the pinch rollers 672, 673 are operated to allow the supply of paper to the camera device under the control of the camera device.

As illustrated in Fig. 164, a mounted silicon chip 53 is insert in one end of the print roll 42. In Fig. 165 the authentication chip 53 is shown in more detail and includes four communications leads 680 - 683 for communicating details from the chip 53 to the corresponding camera to which it is inserted.

Turning to Fig. 165, the chip can be separately created by means of encasing a small integrated circuit 687 in epoxy and running bonding leads eg. 688 to the external communications leads 680 - 683. The integrated chip 687 being approximately 400 microns square with a 100 micron scribe boundary. Subsequently, the chip can be glued to an appropriate surface of the cavity of the print roll 42. In Fig. 166, there is illustrated the integrated circuit 687 interconnected to bonding pads 681, 682 in an exploded view of the arrangement of Fig. 165.

## Authentication Chip

### Authentication Chips 53

The authentication chip 53 of the preferred embodiment is responsible for ensuring that only correctly manufactured print rolls are utilized in the camera system. The authentication chip 53 utilizes technologies that are generally valuable when utilized with any consumables and are not restricted to print roll system. Manufacturers of other systems that require consumables (such as a laser printer that requires toner cartridges) have struggled with the problem of authenticating consumables, to varying levels of success. Most have resorted to specialized packaging. However this does not stop home refill operations or clone manufacture. The prevention of copying is important to prevent poorly manufactured substitute consumables from damaging the base system. For example, poorly filtered ink may clog print nozzles in an ink jet printer, causing the consumer to blame the system manufacturer and not admit the use of non-authorized consumables.

To solve the authentication problem, the Authentication chip 53 contains an authentication code and circuit specially designed to prevent copying. The chip is manufactured using the standard Flash memory manufacturing process, and is low cost enough to be included in consumables such as ink and toner cartridges. Once programmed, the Authentication chips as described here are compliant with the NSA export guidelines. Authentication is an extremely

large and constantly growing field. Here we are concerned with authenticating consumables only.

Symbolic Nomenclature

The following symbolic nomenclature is used throughout the discussion of this embodiment:

| **Symbolic Nomenclature** | **Description** |
|---|---|
| F[X] | Function F, taking a single parameter X |
| F[X, Y] | Function F, taking two parameters, X and Y |
| $X \mid Y$ | X concatenated with Y |
| $X \wedge Y$ | Bitwise X AND Y |
| $X \vee Y$ | Bitwise X OR Y (inclusive-OR) |
| $X \oplus Y$ | Bitwise X XOR Y (exclusive-OR) |
| ~X | Bitwise NOT X (complement) |
| $X \leftarrow Y$ | X is assigned the value Y |
| $X \leftarrow \{Y, Z\}$ | The domain of assignment inputs to X is Y and Z. |
| $X = Y$ | X is equal to Y |
| $X \neq Y$ | X is not equal to Y |
| $\Downarrow X$ | Decrement X by 1 (floor 0) |
| X | Increment X by 1 (with wrapping based on register length) |
| Erase X | Erase Flash memory register X |
| SetBits[X, Y] | Set the bits of the Flash memory register X based on Y |
| $Z \leftarrow$ ShiftRight[X, Y] | Shift register X right one bit position, taking input bit from Y and placing the output bit in Z |

Basic Terms

A message, denoted by **M**, is **plaintext**. The process of transforming M into **cyphertext C**, where the substance of M is hidden, is called **encryption**. The process of transforming C back into M is called **decryption**. Referring to the encryption function as **E**, and the decryption function as **D**, we have the following identities:

$$E[M] = C$$

$$D[C] = M$$

Therefore the following identity is true:

$$D[E[M]] = M$$

Symmetric Cryptography

A symmetric encryption algorithm is one where:

the encryption function E relies on key $K_1$,

the decryption function D relies on key $K_2$,

$K_2$ can be derived from $K_1$, and

$K_1$ can be derived from $K_2$.

In most symmetric algorithms, $K_1$ usually equals $K_2$. However, even if $K_1$ does not equal $K_2$, given that one key can be derived from the other, a single key K can suffice for the mathematical definition. Thus:

$$E_K[M] = C$$

$$D_K[C] = M$$

An enormous variety of symmetric algorithms exist, from the textbooks of ancient history through to sophisticated modern algorithms. Many of these are insecure, in that modern cryptanalysis techniques can successfully attack the algorithm to the extent that K can be derived. The security of the particular symmetric algorithm is normally a function of two things: the strength of the algorithm and the length of the key. The following algorithms include suitable aspects for utilization in the authentication chip.

DES

Blowfish

RC5

IDEA

DES

DES (Data Encryption Standard) is a US and international standard, where the same key is used to encrypt and decrypt. The key length is 56 bits. It has been implemented in hardware and software, although the original design was for hardware only. The original algorithm used in DES is described in US patent 3,962,539. A variant of DES, called triple-DES is more secure, but requires 3 keys: $K_1$, $K_2$, and $K_3$.The keys are used in the following manner:

$$E_{K3}[D_{K2}[E_{K1}[M]]] = C$$

$$D_{K3}[E_{K2}[D_{K1}[C]]] = M$$

The main advantage of triple-DES is that existing DES implementations can be used to give more security than single key DES. Specifically, triple-DES gives protection of equivalent key length of 112 bits. Triple-DES does not give the equivalent protection of a 168-bit key (3 x 56) as one might naively expect. Equipment that performs triple-DES decoding and/or encoding cannot be exported from the United States.

Blowfish

Blowfish, is a symmetric block cipher first presented by Schneier in 1994. It takes a variable length key, from 32 bits to 448 bits. In addition, it is much faster than DES. The Blowfish algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes. Data encryption occurs via a 16-round Feistel network. All operations are XORs and additions on 32-bit words, with four index array lookups per round. It should be noted that decryption is the same as encryption except that the subkey arrays are used in the reverse order. Complexity of implementation is therefore reduced compared to other algorithms that do not have such symmetry.

RC5

Designed by Ron Rivest in 1995, RC5 has a variable block size, key size, and number of rounds. Typically, however, it uses a 64-bit block size and a 128-bit key. The RC5 algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key into 2r+2 subkeys (where r = the number of rounds), each subkey being w bits. For a 64-bit blocksize with 16 rounds (w=32, r=16), the subkey arrays total 136 bytes. Data encryption uses addition mod $2^w$, XOR and bitwise rotation.

IDEA

Developed in 1990 by Lai and Massey, the first incarnation of the IDEA cipher was called PES. After differential cryptanalysis was discovered by Biham and Shamir in 1991, the algorithm was strengthened, with the result being published in 1992 as IDEA. IDEA uses 128 bit-keys to operate on 64-bit plaintext blocks. The same algorithm is used for encryption and decryption. It is generally regarded to be the most secure block algorithm available today. It is described in US Patent No.5,214,703, issued in 1993.

Asymmetric Cryptography

As alternative an asymmetric algorithm could be used. An asymmetric encryption algorithm is one where:

the encryption function E relies on key $K_1$,

the decryption function D relies on key $K_2$,

$K_2$ cannot be derived from $K_1$ in a reasonable amount of time, and

$K_1$ cannot be derived from $K_2$ in a reasonable amount of time.

Thus:

$$E_{K1}[M] = C$$
$$D_{K2}[C] = M$$

These algorithms are also called **public-key** because one key $K_1$ can be made public. Thus anyone can encrypt a message (using $K_1$), but only the person with the corresponding decryption key ($K_2$) can decrypt and thus read the message. In most cases, the following identity also holds:

$$E_{K2}[M] = C$$
$$D_{K1}[C] = M$$

This identity is very important because it implies that anyone with the public key $K_1$ can see M and know that it came from the owner of $K_2$. No-one else could have generated C because to do so would imply knowledge of $K_2$. The property of not being able to derive $K_1$ from $K_2$ and vice versa in a reasonable time is of course clouded by the concept of reasonable time. What has been demonstrated time after time, is that a calculation that was thought to require a long time has been made possible by the introduction of faster computers, new algorithms etc. The security of asymmetric algorithms is based on the difficulty of one of two problems: factoring large numbers (more specifically large numbers that are the product of two large primes), and the difficulty of calculating discrete logarithms in a finite field. Factoring large numbers is conjectured to be a hard problem given today's understanding of mathematics. The problem however, is that factoring is getting easier much faster than anticipated. Ron Rivest in 1977 said that factoring a 125-digit number would take 40 quadrillion years. In 1994 a 129-digit number was factored. According to Schneier, you need a 1024-bit number to get the level of security today that you got from a 512-bit number in the 1980's. If the key is to last for some years then 1024 bits may not even be enough. Rivest revised his key length estimates in 1990: he suggests 1628 bits for high security lasting until 2005, and 1884 bits for high security lasting until 2015. By contrast, Schneier suggests 2048 bits are required in order to protect against corporations and governments until 2015.

A number of public key cryptographic algorithms exist. Most are impractical to implement, and many generate a very large C for a given M or require enormous keys. Still others, while secure, are far too slow to be practical for several years. Because of this, many public-key systems are hybrid – a public key mechanism is used to transmit a symmetric session key, and then the session key is used for the actual messages. All of the algorithms have a problem in terms of key selection. A random number is simply not secure enough. The two large primes p and q must be chosen carefully –

there are certain weak combinations that can be factored more easily (some of the weak keys can be tested for). But nonetheless, key selection is not a simple matter of randomly selecting 1024 bits for example. Consequently the key selection process must also be secure.

Of the practical algorithms in use under public scrutiny, the following may be suitable for utilization:

RSA

DSA

ElGamal

RSA

The RSA cryptosystem, named after Rivest, Shamir, and Adleman, is the most widely used public-key cryptosystem, and is a de facto standard in much of the world. The security of RSA is conjectured to depend on the difficulty of factoring large numbers that are the product of two primes (p and q). There are a number of restrictions on the generation of p and q. They should both be large, with a similar number of bits, yet not be close to one another (otherwise $pq \approx \sqrt{pq}$). In addition, many authors have suggested that p and q should be strong primes. The RSA algorithm patent was issued in 1983 (US patent number 4,405,829).

DSA

DSA (Digital Signature Standard) is an algorithm designed as part of the Digital Signature Standard (DSS). As defined, it cannot be used for generalized encryption. In addition, compared to RSA, DSA is 10 to 40 times slower for signature verification. DSA explicitly uses the SHA-1 hashing algorithm (see definition in

**One-way** Functions below). DSA key generation relies on finding two primes p and q such that q divides p-1. According to Schneier, a 1024-bit p value is required for long term DSA security. However the DSA standard does not permit values of p larger than 1024 bits (p must also be a multiple of 64 bits). The US Government owns the DSA algorithm and has at least one relevant patent (US patent 5,231,688 granted in 1993).

ElGamal

The ElGamal scheme is used for both encryption and digital signatures. The security is based on the difficulty of calculating discrete logarithms in a finite field. Key selection involves the selection of a prime p, and two random numbers g and x such that both g and x are less than p. Then calculate y = gx mod p. The public key is y, g, and p. The private key is x.

Cryptographic Challenge-Response Protocols and Zero Knowledge Proofs

The general principle of a challenge-response protocol is to provide identity authentication adapted to a camera system. The simplest form of challenge-response takes the form of a secret password. A asks B for the secret password, and if B responds with the correct password, A declares B authentic. There are three main problems with this kind of simplistic protocol. Firstly, once B has given out the password, any observer C will know what the password is. Secondly, A must know the password in order to verify it. Thirdly, if C impersonates A, then B will give the password to C (thinking C was A), thus compromising B. Using a copyright text (such as a haiku) is a weaker alternative as we are assuming that anyone is able to copy the password (for example in a country where intellectual property is not respected). The idea of cryptographic challenge-response protocols is that one entity (the claimant) proves its identity to another (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, without revealing the secret itself to the verifier during the protocol. In the generalized case of cryptographic challenge-

response protocols, with some schemes the verifier knows the secret, while in others the secret is not even known by the verifier. Since the discussion of this embodiment specifically concerns Authentication, the actual cryptographic challenge-response protocols used for authentication are detailed in the appropriate sections. However the concept of Zero Knowledge Proofs will be discussed here. The Zero Knowledge Proof protocol, first described by Feige, Fiat and Shamir is extensively used in Smart Cards for the purpose of authentication. The protocol's effectiveness is based on the assumption that it is computationally infeasible to compute square roots modulo a large composite integer with unknown factorization. This is provably equivalent to the assumption that factoring large integers is difficult. It should be noted that there is no need for the claimant to have significant computing power. Smart cards implement this kind of authentication using only a few modular multiplications. The Zero Knowledge Proof protocol is described in US Patent 4,748,668.

One-way Functions

A one-way function F operates on an input X, and returns F[X] such that X cannot be determined from F[X]. When there is no restriction on the format of X, and F[X] contains fewer bits than X, then collisions must exist. A collision is defined as two different X input values producing the same F[X] value - i.e. $X_1$ and $X_2$ exist such that $X_1$ $X_2$ yet $F[X_1] = F[X_2]$. When X contains more bits than F[X], the input must be compressed in some way to create the output. In many cases, X is broken into blocks of a particular size, and compressed over a number of rounds, with the output of one round being the input to the next. The output of the hash function is the last output once X has been consumed. A pseudo-collision of the compression function CF is defined as two different initial values $V_1$ and $V_2$ and two inputs $X_1$ and $X_2$ (possibly identical) are given such that $CF(V_1, X_1) = CF(V_2, X_2)$. Note that the existence of a pseudo-collision does not mean that it is easy to compute an $X_2$ for a given $X_1$.

We are only interested in one-way functions that are fast to compute. In addition, we are only interested in deterministic one-way functions that are repeatable in different implementations. Consider an example F where F[X] is the time between calls to F. For a given F[X] X cannot be determined because X is not even used by F. However the output from F will be different for different implementations. This kind of F is therefore not of interest.

In the scope of the discussion of the implementation of the authentication chip of this embodiment, we are interested in the following forms of one-way functions:

- Encryption using an unknown key
- Random number sequences
- Hash Functions
- Message Authentication Codes

Encryption Using an Unknown Key

When a message is encrypted using an unknown key K, the encryption function E is effectively one-way. Without the key, it is computationally infeasible to obtain M from $E_K[M]$ without K. An encryption function is only one-way for as long as the key remains hidden. An encryption algorithm does not create collisions, since E creates $E_K[M]$ such that it is possible to reconstruct M using function D. Consequently F[X] contains at least as many bits as X (no information is lost) if the one-way function F is E. Symmetric encryption algorithms (see above) have the advantage over Asymmetric algorithms for producing one-way functions based on encryption for the following reasons:

The key for a given strength encryption algorithm is shorter for a symmetric algorithm than an asymmetric algorithm

Symmetric algorithms are faster to compute and require less software/silicon

The selection of a good key depends on the encryption algorithm chosen. Certain keys are not strong for particular encryption algorithms, so any key needs to be tested for strength. The more tests that need to be performed for key selection, the less likely the key will remain hidden.

Random Number Sequences

Consider a random number sequence $R_0, R_1, \ldots, R_I, R_{i+1}$. We define the one-way function F such that F[X] returns the $X^{th}$ random number in the random sequence. However we must ensure that F[X] is repeatable for a given X on different implementations. The random number sequence therefore cannot be truly random. Instead, it must be pseudo-random, with the generator making use of a specific seed.

There are a large number of issues concerned with defining good random number generators. Knuth, describes what makes a generator "good" (including statistical tests), and the general problems associated with constructing them. The majority of random number generators produce the $i^{th}$ random number from the $i$-$1^{th}$ state – the only way to determine the $i^{th}$ number is to iterate from the $0^{th}$ number to the $i^{th}$. If i is large, it may not be practical to wait for i iterations. However there is a type of random number generator that does allow random access. Blum, Blum and Shub define the ideal generator as follows:"... we would like a pseudo-random sequence generator to quickly produce, from short seeds, long sequences (of bits) that appear in every way to be generated by successive flips of a fair coin". They defined the ${}^{2}$ **mod n** generator, more commonly referred to as the BBS generator. They showed that given certain assumptions upon which modern cryptography relies, a BBS generator passes extremely stringent statistical tests.

The BBS generator relies on selecting n which is a Blum integer (n = pq where p and q are large prime numbers, $p \neq q$, p mod 4 = 3, and q mod 4 = 3). The initial state of the generator is given by ${}_0$ where ${}_0 = {}^2$ mod n, and x is a random integer relatively prime to n. The $i^{th}$ pseudo-random bit is the least significant bit of $x_i$ where $x_i = x_{i-1}^2$ mod n. As an extra property, knowledge of p and q allows a direct calculation of the $i^{th}$ number in the sequence as follows: $x_i = {}_0^{y}$ mod n, where $y = 2^i$ mod ((p-1)(q-1))

Without knowledge of p and q, the generator must iterate (the security of calculation relies on the difficulty of factoring large numbers). When first defined, the primary problem with the BBS generator was the amount of work required for a single output bit. The algorithm was considered too slow for most applications. However the advent of Montgomery reduction arithmetic has given rise to more practical implementations. In addition, Vazirani and Vazirani have shown that depending on the size of n, more bits can safely be taken from $x_i$ without compromising the security of the generator. Assuming we only take 1 bit per $x_i$, N bits (and hence N iterations of the bit generator function) are needed in order to generate an N-bit random number. To the outside observer, given a particular set of bits, there is no way to determine the next bit other than a 50/50 probability. If the x, p and q are hidden, they act as a key, and it is computationally unfeasible to take an output bit stream and compute x, p, and q. It is also computationally unfeasible to determine the value of i used to generate a given set of pseudo-random bits. This last feature makes the generator one-way. Different values of i can produce identical bit sequences of a given length (e.g. 32 bits of random bits). Even if x, p and q are known, for a given F[i], i can only be derived as a set of possibilities, not as a certain value (of course if the domain of i is known, then the set of possibilities is reduced further). However, there are problems in selecting a good p and q, and a good seed x. In particular, Ritter describes a problem in selecting x. The nature of the problem is that a

BBS generator does not create a single cycle of known length. Instead, it creates cycles of various lengths, including degenerate (zero-length) cycles. Thus a BBS generator cannot be initialized with a random state – it might be on a short cycle.

Hash Functions

Special one-way functions, known as Hash functions map arbitrary length messages to fixed-length hash values. Hash functions are referred to as H[M]. Since the input is arbitrary length, a hash function has a compression component in order to produce a fixed length output. Hash functions also have an obfuscation component in order to make it difficult to find collisions and to determine information about M from H[M]. Because collisions do exist, most applications require that the hash algorithm is preimage resistant, in that for a given $X_1$ it is difficult to find $X_2$ such that $H[X_1] = H[X_2]$. In addition, most applications also require the hash algorithm to be collision resistant (i.e. it should be hard to find two messages $X_1$ and $X_2$ such that $H[X_1] = H[X_2]$). It is an open problem whether a collision-resistant hash function, in the idealist sense, can exist at all. The primary application for hash functions is in the reduction of an input message into a digital "fingerprint" before the application of a digital signature algorithm. One problem of collisions with digital signatures can be seen in the following example.

> A has a long message $M_1$ that says "I owe B $10". A signs $H[M_1]$ using his private key. B, being greedy, then searches for a collision message $M_2$ where $H[M_2] = H[M_1]$ but where $M_2$ is favorable to B, for example "I owe B $1million". Clearly it is in A's interest to ensure that it is difficult to find such an $M_2$.

Examples of collision resistant one-way hash functions are SHA-1, MD5 and RIPEMD-160, all derived from MD4.

MD4

Ron Rivest introduced MD4 in 1990. It is mentioned here because all other one-way hash functions are derived in some way from MD4. MD4 is now considered completely broken in that collisions can be calculated instead of searched for. In the example above, B could trivially generate a substitute message $M_2$ with the same hash value as the original message $M_1$.

MD5

Ron Rivest introduced MD5 in 1991 as a more secure MD4. Like MD4, MD5 produces a 128-bit hash value. Dobbertin describes the status of MD5 after recent attacks. He describes how pseudo-collisions have been found in MD5, indicating a weakness in the compression function, and more recently, collisions have been found. This means that MD5 should not be used for compression in digital signature schemes where the existence of collisions may have dire consequences. However MD5 can still be used as a one-way function. In addition, the HMAC-MD5 construct is not affected by these recent attacks.

SHA-1

SHA-1 is very similar to MD5, but has a 160-bit hash value (MD5 only has 128 bits of hash value). SHA-1 was designed and introduced by the NIST and NSA for use in the Digital Signature Standard (DSS). The original published description was called SHA, but very soon afterwards, was revised to become SHA-1, supposedly to correct a security flaw in SHA (although the NSA has not released the mathematical reasoning behind the change). There are no known cryptographic attacks against SHA-1. It is also more resistant to brute-force attacks than MD4 or MD5 simply because of the longer hash result. The US Government owns the SHA-1 and DSA algorithms (a digital signature authentication algorithm defined as part of DSS) and has at least one relevant patent (US patent 5,231,688 granted in

1993).

RIPEMD-160

RIPEMD-160 is a hash function derived from its predecessor RIPEMD (developed for the European Community's RIPE project in 1992). As its name suggests, RIPEMD-160 produces a 160-bit hash result. Tuned for software implementations on 32-bit architectures, RIPEMD-160 is intended to provide a high level of security for 10 years or more. Although there have been no successful attacks on RIPEMD-160, it is comparatively new and has not been extensively cryptanalyzed. The original RIPEMD algorithm was specifically designed to resist known cryptographic attacks on MD4. The recent attacks on MD5 showed similar weaknesses in the RIPEMD 128-bit hash function. Although the attacks showed only theoretical weaknesses, Dobbertin, Preneel and Bosselaers further strengthened RIPEMD into a new algorithm RIPEMD-160.

Message Authentication Codes

The problem of message authentication can be summed up as follows:

How can A be sure that a message supposedly from B is in fact from B?

Message authentication is different from entity authentication. With entity authentication, one entity (the claimant) proves its identity to another (the verifier). With message authentication, we are concerned with making sure that a given message is from who we think it is from i.e. it has not been tampered en route from the source to its destination. A one-way hash function is not sufficient protection for a message. Hash functions such as MD5 rely on generating a hash value that is representative of the original input, and the original input cannot be derived from the hash value. A simple attack by E, who is in-between A and B, is to intercept the message from B, and substitute his own. Even if A also sends a hash of the original message, E can simply substitute the hash of his new message. Using a one-way hash function alone, A has no way of knowing that B's message has been changed. One solution to the problem of message authentication is the Message Authentication Code, or MAC. When B sends message M, it also sends MAC[M] so that the receiver will know that M is actually from B. For this to be possible, only B must be able to produce a MAC of M, and in addition, A should be able to verify M against MAC[M]. Notice that this is different from encryption of M - MACs are useful when M does not have to be secret. The simplest method of constructing a MAC from a hash function is to encrypt the hash value with a symmetric algorithm:

Hash the input message H[M]

Encrypt the hash $E_K[H[M]]$

This is more secure than first encrypting the message and then hashing the encrypted message. Any symmetric or asymmetric cryptographic function can be used. However, there are advantages to using a key-dependant one-way hash function instead of techniques that use encryption (such as that shown above):

- Speed, because one-way hash functions in general work much faster than encryption;
- Message size, because $E_K[H[M]]$ is at least the same size as M, while H[M] is a fixed size (usually considerably smaller than M);
- Hardware/software requirements – keyed one-way hash functions are typically far less complexity than their encryption-based counterparts; and
- One-way hash function implementations are not considered to be encryption or decryption devices and therefore are not subject to US export controls.

It should be noted that hash functions were never originally designed to contain a key or to support message

authentication. As a result, some ad hoc methods of using hash functions to perform message authentication, including various functions that concatenate messages with secret prefixes, suffixes, or both have been proposed. Most of these ad hoc methods have been successfully attacked by sophisticated means. Additional MACs have been suggested based on XOR schemes and Toeplitz matricies (including the special case of LFSR-based constructions).

HMAC

The HMAC construction in particular is gaining acceptance as a solution for Internet message authentication security protocols. The HMAC construction acts as a wrapper, using the underlying hash function in a black-box way. Replacement of the hash function is straightforward if desired due to security or performance reasons. However, the major advantage of the HMAC construct is that it can be proven secure provided the underlying hash function has some reasonable cryptographic strengths – that is, HMAC's strengths are directly connected to the strength of the hash function. Since the HMAC construct is a wrapper, any iterative hash function can be used in an HMAC. Examples include HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD160 etc. Given the following definitions:

| | | |
|---|---|---|
| H | = | the hash function (e.g. MD5 or SHA-1) |
| n | = | number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5) |
| M | = | the data to which the MAC function is to be applied |
| K | = | the secret key shared by the two parties |
| ipad | = | 0x36 repeated 64 times |
| opad | = | 0x5C repeated 64 times |

The HMAC algorithm is as follows:

Extend K to 64 bytes by appending 0x00 bytes to the end of K

XOR the 64 byte string created in (1) with ipad

Append data stream M to the 64 byte string created in (2)

Apply H to the stream generated in (3)

XOR the 64 byte string created in (1) with opad

Append the H result from (4) to the 64 byte string resulting from (5)

Apply H to the output of (6) and output the result

Thus:

$$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad)|M]]$$

The recommended key length is at least n bits, although it should not be longer than 64 bytes (the length of the hashing block). A key longer than n bits does not add to the security of the function. HMAC optionally allows truncation of the final output e.g. truncation to 128 bits from 160 bits. The HMAC designers' Request for Comments was issued in 1997, one year after the algorithm was first introduced. The designers claimed that the strongest known attack against HMAC is based on the frequency of collisions for the hash function H and is totally impractical for minimally reasonable hash functions. More recently, HMAC protocols with replay prevention components have been defined in order to prevent the capture and replay of any M, HMAC[M] combination within a given time period.

Random Numbers and Time Varying Messages

The use of a random number generator as a one-way function has already been examined. However, random number generator theory is very much intertwined with cryptography, security, and authentication. There are a large number of issues concerned with defining good random number generators. Knuth, describes what makes a generator good

(including statistical tests), and the general problems associated with constructing them. One of the uses for random numbers is to ensure that messages vary over time. Consider a system where A encrypts commands and sends them to B. If the encryption algorithm produces the same output for a given input, an attacker could simply record the messages and play them back to fool B. There is no need for the attacker to crack the encryption mechanism other than to know which message to play to B (while pretending to be A). Consequently messages often include a random number and a time stamp to ensure that the message (and hence its encrypted counterpart) varies each time. Random number generators are also often used to generate keys. It is therefore best to say at the moment, that all generators are insecure for this purpose. For example, the Berlekamp-Massey algorithm, is a classic attack on an LFSR random number generator. If the LFSR is of length n, then only 2n bits of the sequence suffice to determine the LFSR, compromising the key generator. If, however, the only role of the random number generator is to make sure that messages vary over time, the security of the generator and seed is not as important as it is for session key generation. If however, the random number seed generator is compromised, and an attacker is able to calculate future "random" numbers, it can leave some protocols open to attack. Any new protocol should be examined with respect to this situation. The actual type of random number generator required will depend upon the implementation and the purposes for which the generator is used. Generators include Blum, Blum, and Shub, stream ciphers such as RC4 by Ron Rivest, hash functions such as SHA-1 and RIPEMD-160, and traditional generators such LFSRs (Linear Feedback Shift Registers) and their more recent counterpart FCSRs (Feedback with Carry Shift Registers).

Attacks

This section describes the various types of attacks that can be undertaken to break an authentication cryptosystem such as the authentication chip. The attacks are grouped into **physical** and **logical** attacks. Physical attacks describe methods for breaking a physical implementation of a cryptosystem (for example, breaking open a chip to retrieve the key), while logical attacks involve attacks on the cryptosystem that are implementation independent. Logical types of attack work on the protocols or algorithms, and attempt to do one of three things:

- Bypass the authentication process altogether
- Obtain the secret key by force or deduction, so that any question can be answered
- Find enough about the nature of the authenticating questions and answers in order to, without the key, give the right answer to each question.

The attack styles and the forms they take are detailed below. Regardless of the algorithms and protocol used by a security chip, the circuitry of the authentication part of the chip can come under physical attack. Physical attack comes in four main ways, although the form of the attack can vary:

- Bypassing the Authentication Chip altogether
- Physical examination of chip while in operation (destructive and non-destructive)
- Physical decomposition of chip
- Physical alteration of chip

The attack styles and the forms they take are detailed below. This section does not suggest solutions to these attacks. It merely describes each attack type. The examination is restricted to the context of an Authentication chip 53 (as opposed to some other kind of system, such as Internet authentication) attached to some System.

Logical Attacks

These attacks are those which do not depend on the physical implementation of the cryptosystem. They work against

the protocols and the security of the algorithms and random number generators.

Ciphertext only attack

This is where an attacker has one or more encrypted messages, all encrypted using the same algorithm. The aim of the attacker is to obtain the plaintext messages from the encrypted messages. Ideally, the key can be recovered so that all messages in the future can also be recovered.

Known plaintext attack

This is where an attacker has both the plaintext and the encrypted form of the plaintext. In the case of an Authentication Chip, a known-plaintext attack is one where the attacker can see the data flow between the System and the Authentication Chip. The inputs and outputs are observed (not chosen by the attacker), and can be analyzed for weaknesses (such as birthday attacks or by a search for differentially interesting input/output pairs). A known plaintext attack is a weaker type of attack than the chosen plaintext attack, since the attacker can only observe the data flow. A known plaintext attack can be carried out by connecting a logic analyzer to the connection between the System and the Authentication Chip.

Chosen plaintext attacks

A chosen plaintext attack describes one where a cryptanalyst has the ability to send any chosen message to the cryptosystem, and observe the response. If the cryptanalyst knows the algorithm, there may be a relationship between inputs and outputs that can be exploited by feeding a specific output to the input of another function. On a system using an embedded Authentication Chip, it is generally very difficult to prevent chosen plaintext attacks since the cryptanalyst can logically pretend he/she is the System, and thus send any chosen bit-pattern streams to the Authentication Chip.

Adaptive Chosen plaintext attacks

This type of attack is similar to the chosen plaintext attacks except that the attacker has the added ability to modify subsequent chosen plaintexts based upon the results of previous experiments. This is certainly the case with any System / Authentication Chip scenario described when utilized for consumables such as photocopiers and toner cartridges, especially since both Systems and Consumables are made available to the public.

Brute force attack

A guaranteed way to break any key-based cryptosystem algorithm is simply to try every key. Eventually the right one will be found. This is known as a **Brute Force Attack**. However, the more key possibilities there are, the more keys must be tried, and hence the longer it takes (on average) to find the right one. If there are N keys, it will take a maximum of N tries. If the key is N bits long, it will take a maximum of $2^N$ tries, with a 50% chance of finding the key after only half the attempts ($2^{N-1}$). The longer N becomes, the longer it will take to find the key, and hence the more secure the key is. Of course, an attack may guess the key on the first try, but this is more unlikely the longer the key is. Consider a key length of 56 bits. In the worst case, all $2^{56}$ tests ($7.2 \times 10^{16}$ tests) must be made to find the key. In 1977, Diffie and Hellman described a specialized machine for cracking DES, consisting of one million processors, each capable of running one million tests per second. Such a machine would take 20 hours to break any DES code. Consider a key length of 128 bits. In the worst case, all $2^{128}$ tests ($3.4 \times 10^{38}$ tests) must be made to find the key. This would take ten billion years on an array of a trillion processors each running 1 billion tests per second. With a long enough key length, a Brute Force Attack takes too long to be worth the attacker's efforts.

Guessing attack

This type of attack is where an attacker attempts to simply "guess" the key. As an attack it is identical to the Brute force attack, where the odds of success depend on the length of the key.

Quantum Computer attack

To break an n-bit key, a quantum computer (NMR, Optical, or Caged Atom) containing n qubits embedded in an appropriate algorithm must be built. The quantum computer effectively exists in $2^n$ simultaneous coherent states. The trick is to extract the right coherent state without causing any decoherence. To date this has been achieved with a 2 qubit system (which exists in 4 coherent states). It is thought possible to extend this to 6 qubits (with 64 simultaneous coherent states) within a few years.

Unfortunately, every additional qubit halves the relative strength of the signal representing the key. This rapidly becomes a serious impediment to key retrieval, especially with the long keys used in cryptographically secure systems. As a result, attacks on a cryptographically secure key (e.g. 160 bits) using a Quantum Computer are likely not to be feasible and it is extremely unlikely that quantum computers will have achieved more than 50 or so qubits within the commercial lifetime of the Authentication Chips. Even using a 50 qubit quantum computer, $2^{110}$ tests are required to crack a 160 bit key.

Purposeful Error Attack

With certain algorithms, attackers can gather valuable information from the results of a bad input. This can range from the error message text to the time taken for the error to be generated. A simple example is that of a userid/password scheme. If the error message usually says "Bad userid", then when an attacker gets a message saying "Bad password" instead, then they know that the userid is correct. If the message always says "Bad userid/password" then much less information is given to the attacker. A more complex example is that of the recent published method of cracking encryption codes from secure web sites. The attack involves sending particular messages to a server and observing the error message responses. The responses give enough information to learn the keys – even the lack of a response gives some information. An example of algorithmic time can be seen with an algorithm that returns an error as soon as an erroneous bit is detected in the input message. Depending on hardware implementation, it may be a simple method for the attacker to time the response and alter each bit one by one depending on the time taken for the error response, and thus obtain the key. Certainly in a chip implementation the time taken can be observed with far greater accuracy than over the Internet.

Birthday attack

This attack is named after the famous "birthday paradox" (which is not actually a paradox at all). The odds of one person sharing a birthday with another, is 1 in 365 (not counting leap years). Therefore there must be 183 people in a room for the odds to be more than 50% that one of them shares your birthday. However, there only needs to be 23 people in a room for there to be more than a 50% chance that any two share a birthday. This is because 23 people yields 253 different pairs. Birthday attacks are common attacks against hashing algorithms, especially those algorithms that combine hashing with digital signatures. If a message has been generated and already signed, an attacker must search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday). However, if the attacker can generate the message, the Birthday Attack comes into play. The attacker searches for two messages that share the same hash value (analogous to any two people sharing a birthday), only one message is acceptable to the person signing it, and the other is beneficial for the attacker. Once the person has signed the original message the attacker simply claims now that the person signed the alternative message – mathematically

there is no way to tell which message was the original, since they both hash to the same value. Assuming a Brute Force Attack is the only way to determine a match, the weakening of an n-bit key by the birthday attack is $2^{n/2}$. A key length of 128 bits that is susceptible to the birthday attack has an effective length of only 64 bits.

Chaining attack

These are attacks made against the chaining nature of hash functions. They focus on the compression function of a hash function. The idea is based on the fact that a hash function generally takes arbitrary length input and produces a constant length output by processing the input n bits at a time. The output from one block is used as the chaining variable set into the next block. Rather than finding a collision against an entire input, the idea is that given an input chaining variable set, to find a substitute block that will result in the same output chaining variables as the proper message. The number of choices for a particular block is based on the length of the block. If the chaining variable is c bits, the hashing function behaves like a random mapping, and the block length is b bits, the number of such b-bit blocks is approximately 2b / 2c. The challenge for finding a substitution block is that such blocks are a sparse subset of all possible blocks. For SHA-1, the number of 512 bit blocks is approximately $2^{512}/2^{160}$, or $2^{352}$. The chance of finding a block by brute force search is about 1 in $2^{160}$.

Substitution with a complete lookup table

If the number of potential messages sent to the chip is small, then there is no need for a clone manufacturer to crack the key. Instead, the clone manufacturer could incorporate a ROM in their chip that had a record of all of the responses from a genuine chip to the codes sent by the system. The larger the key, and the larger the response, the more space is required for such a lookup table.

Substitution with a sparse lookup table

If the messages sent to the chip are somehow predictable, rather than effectively random, then the clone manufacturer need not provide a complete lookup table. For example:

If the message is simply a serial number, the clone manufacturer need simply provide a lookup table that contains values for past and predicted future serial numbers. There are unlikely to be more than $10^9$ of these.

If the test code is simply the date, then the clone manufacturer can produce a lookup table using the date as the address.

If the test code is a pseudo-random number using either the serial number or the date as a seed, then the clone manufacturer just needs to crack the pseudo-random number generator in the System. This is probably not difficult, as they have access to the object code of the System. The clone manufacturer would then produce a content addressable memory (or other sparse array lookup) using these codes to access stored authentication codes.

Differential cryptanalysis

Differential cryptanalysis describes an attack where pairs of input streams are generated with known differences, and the differences in the encoded streams are analyzed. Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although other algorithms such as HMAC-SHA1 have no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

Minimal-difference inputs, and their corresponding outputs

Minimal-difference outputs, and their corresponding inputs

Most algorithms were strengthened against differential cryptanalysis once the process was described. This is covered in

the specific sections devoted to each cryptographic algorithm. However some recent algorithms developed in secret have been broken because the developers had not considered certain styles of differential attacks and did not subject their algorithms to public scrutiny.

Message substitution attacks

In certain protocols, a man-in-the-middle can substitute part or all of a message. This is where a real Authentication Chip is plugged into a reusable clone chip within the consumable. The clone chip intercepts all messages between the System and the Authentication Chip, and can perform a number of substitution attacks. Consider a message containing a header followed by content. An attacker may not be able to generate a valid header, but may be able to substitute their own content, especially if the valid response is something along the lines of "Yes, I received your message". Even if the return message is "Yes, I received the following message ...", the attacker may be able to substitute the original message before sending the acknowledgement back to the original sender. Message Authentication Codes were developed to combat most message substitution attacks.

Reverse engineering the key generator

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the Netscape security program was initially broken.

Bypassing authentication altogether

It may be that there are problems in the authentication protocols that can allow a bypass of the authentication process altogether. With these kinds of attacks the key is completely irrelevant, and the attacker has no need to recover it or deduce it. Consider an example of a system that Authenticates at power-up, but does not authenticate at any other time. A reusable consumable with a clone Authentication Chip may make use of a real Authentication Chip. The clone authentication chip 53 uses the real chip for the authentication call, and then simulates the real Authentication Chip's state data after that. Another example of bypassing authentication is if the System authenticates only after the consumable has been used. A clone Authentication Chip can accomplish a simple authentication bypass by simulating a loss of connection after the use of the consumable but before the authentication protocol has completed (or even started). One infamous attack known as the "Kentucky Fried Chip" hack involved replacing a microcontroller chip for a satellite TV system. When a subscriber stopped paying the subscription fee, the system would send out a "disable" message. However the new microcontroller would simply detect this message and not pass it on to the consumer's satellite TV system.

Garrote/bribe attack

If people know the key, there is the possibility that they could tell someone else. The telling may be due to coercion (bribe, garrote etc), revenge (e.g. a disgruntled employee), or simply for principle. These attacks are usually cheaper and easier than other efforts at deducing the key. As an example, a number of people claiming to be involved with the development of the Divx standard have recently (May/June 1998) been making noises on a variety of DVD newsgroups to the effect they would like to help develop Divx specific cracking devices – out of principle.

Physical Attacks

The following attacks assume implementation of an authentication mechanism in a silicon chip that the attacker has physical access to. The first attack, **Reading ROM**, describes an attack when keys are stored in ROM, while the remaining attacks assume that a secret key is stored in Flash memory.

Reading ROM

If a key is stored in ROM it can be read directly. A ROM can thus be safely used to hold a public key (for use in asymmetric cryptography), but not to hold a private key. In symmetric cryptography, a ROM is completely insecure. Using a copyright text (such as a haiku) as the key is not sufficient, because we are assuming that the cloning of the chip is occurring in a country where intellectual property is not respected.

Reverse engineering of chip

Reverse engineering of the chip is where an attacker opens the chip and analyzes the circuitry. Once the circuitry has been analyzed the inner workings of the chip's algorithm can be recovered. Lucent Technologies have developed an active method known as TOBIC (Two photon OBIC, where OBIC stands for Optical Beam Induced Current), to image circuits. Developed primarily for static RAM analysis, the process involves removing any back materials, polishing the back surface to a mirror finish, and then focusing light on the surface. The excitation wavelength is specifically chosen not to induce a current in the IC. A Kerckhoffs in the nineteenth century made a fundamental assumption about cryptanalysis: if the algorithm's inner workings are the sole secret of the scheme, the scheme is as good as broken. He stipulated that the secrecy must reside entirely in the key. As a result, the best way to protect against reverse engineering of the chip is to make the inner workings irrelevant.

Usurping the authentication process

It must be assumed that any clone manufacturer has access to both the System and consumable designs. If the same channel is used for communication between the System and a trusted System Authentication Chip, and a non-trusted consumable Authentication Chip, it may be possible for the non-trusted chip to interrogate a trusted Authentication Chip in order to obtain the "correct answer". If this is so, a clone manufacturer would not have to determine the key. They would only have to trick the System into using the responses from the System Authentication Chip. The alternative method of usurping the authentication process follows the same method as the logical attack "Bypassing the Authentication Process", involving simulated loss of contact with the System whenever authentication processes take place, simulating power-down etc.

Modification of System

This kind of attack is where the System itself is modified to accept clone consumables. The attack may be a change of System ROM, a rewiring of the consumable, or, taken to the extreme case, a completely clone System. This kind of attack requires each individual System to be modified, and would most likely require the owner's consent. There would usually have to be a clear advantage for the consumer to undertake such a modification, since it would typically void warranty and would most likely be costly. An example of such a modification with a clear advantage to the consumer is a software patch to change fixed-region DVD players into region-free DVD players.

Direct viewing of chip operation by conventional probing

If chip operation could be directly viewed using an STM or an electron beam, the keys could be recorded as they are read from the internal non-volatile memory and loaded into work registers. These forms of conventional probing require direct access to the top or front sides of the IC while it is powered.

Direct viewing of the non-volatile memory

If the chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the key could probably be viewed directly using an STM or SKM (Scanning Kelvin Microscope). However, slicing the chip to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling

(focused ion beam etching), or chemical mechanical polishing will almost certainly discharge the small charges present on the floating gates.

Viewing the light bursts caused by state changes

Whenever a gate changes state, a small amount of infrared energy is emitted. Since silicon is transparent to infrared, these changes can be observed by looking at the circuitry from the underside of a chip. While the emission process is weak, it is bright enough to be detected by highly sensitive equipment developed for use in astronomy. The technique, developed by IBM, is called PICA (Picosecond Imaging Circuit Analyzer). If the state of a register is known at time t, then watching that register change over time will reveal the exact value at time t+n, and if the data is part of the key, then that part is compromised.

Monitoring EMI

Whenever electronic circuitry operates, faint electromagnetic signals are given off. Relatively inexpensive equipment (a few thousand dollars) can monitor these signals. This could give enough information to allow an attacker to deduce the keys.

Viewing $I_{dd}$ fluctuations

Even if keys cannot be viewed, there is a fluctuation in current whenever registers change state. If there is a high enough signal to noise ratio, an attacker can monitor the difference in $I_{dd}$ that may occur when programming over either a high or a low bit. The change in $I_{dd}$ can reveal information about the key. Attacks such as these have already been used to break smart cards.

Differential Fault Analysis

This attack assumes introduction of a bit error by ionization, microwave radiation, or environmental stress. In most cases such an error is more likely to adversely affect the Chip (eg cause the program code to crash) rather than cause beneficial changes which would reveal the key. Targeted faults such as ROM overwrite, gate destruction etc are far more likely to produce useful results.

Clock glitch attacks

Chips are typically designed to properly operate within a certain clock speed range. Some attackers attempt to introduce faults in logic by running the chip at extremely high clock speeds or introduce a clock glitch at a particular time for a particular duration. The idea is to create race conditions where the circuitry does not function properly. An example could be an AND gate that (because of race conditions) gates through $Input_1$ all the time instead of the AND of $Input_1$ and $Input_2$. If an attacker knows the internal structure of the chip, they can attempt to introduce race conditions at the correct moment in the algorithm execution, thereby revealing information about the key (or in the worst case, the key itself).

Power supply attacks

Instead of creating a glitch in the clock signal, attackers can also produce glitches in the power supply where the power is increased or decreased to be outside the working operating voltage range. The net effect is the same as a clock glitch – introduction of error in the execution of a particular instruction. The idea is to stop the CPU from XORing the key, or from shifting the data one bit-position etc. Specific instructions are targeted so that information about the key is revealed.

Overwriting ROM

Single bits in a ROM can be overwritten using a laser cutter microscope, to either 1 or 0 depending on the sense of the

logic. With a given opcode/operand set, it may be a simple matter for an attacker to change a conditional jump to a non-conditional jump, or perhaps change the destination of a register transfer. If the target instruction is chosen carefully, it may result in the key being revealed.

Modifying EEPROM/Flash

EEPROM/Flash attacks are similar to ROM attacks except that the laser cutter microscope technique can be used to both set and reset individual bits. This gives much greater scope in terms of modification of algorithms.

Gate Destruction

Anderson and Kuhn described the rump session of the 1997 workshop on Fast Software Encryption, where Biham and Shamir presented an attack on DES. The attack was to use a laser cutter to destroy an individual gate in the hardware implementation of a known block cipher (DES). The net effect of the attack was to force a particular bit of a register to be "stuck". Biham and Shamir described the effect of forcing a particular register to be affected in this way – the least significant bit of the output from the round function is set to 0. Comparing the 6 least significant bits of the left half and the right half can recover several bits of the key. Damaging a number of chips in this way can reveal enough information about the key to make complete key recovery easy. An encryption chip modified in this way will have the property that encryption and decryption will no longer be inverses.

Overwrite Attacks

Instead of trying to read the Flash memory, an attacker may simply set a single bit by use of a laser cutter microscope. Although the attacker doesn't know the previous value, they know the new value. If the chip still works, the bit's original state must be the same as the new state. If the chip doesn't work any longer, the bit's original state must be the logical NOT of the current state. An attacker can perform this attack on each bit of the key and obtain the n-bit key using at most n chips (if the new bit matched the old bit, a new chip is not required for determining the next bit).

Test Circuitry Attack

Most chips contain test circuitry specifically designed to check for manufacturing defects. This includes BIST (Built In Self Test) and scan paths. Quite often the scan paths and test circuitry includes access and readout mechanisms for all the embedded latches. In some cases the test circuitry could potentially be used to give information about the contents of particular registers. Test circuitry is often disabled once the chip has passed all manufacturing tests, in some cases by blowing a specific connection within the chip. A determined attacker, however, can reconnect the test circuitry and hence enable it.

Memory Remanence

Values remain in RAM long after the power has been removed, although they do not remain long enough to be considered non-volatile. An attacker can remove power once sensitive information has been moved into RAM (for example working registers), and then attempt to read the value from RAM. This attack is most useful against security systems that have regular RAM chips. A classic example is where a security system was designed with an automatic power-shut-off that is triggered when the computer case is opened. The attacker was able to simply open the case, remove the RAM chips, and retrieve the key because of memory remanence.

Chip Theft Attack

If there are a number of stages in the lifetime of an Authentication Chip, each of these stages must be examined in terms of ramifications for security should chips be stolen. For example, if information is programmed into the chip in stages, theft of a chip between stages may allow an attacker to have access to key information or reduced efforts for

attack. Similarly, if a chip is stolen directly after manufacture but before programming, does it give an attacker any logical or physical advantage?

Requirements

Existing solutions to the problem of authenticating consumables have typically relied on physical patents on packaging. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. Consequently a much higher level of protection is required. The authentication mechanism is therefore built into an Authentication chip 53 that allows a system to authenticate a consumable securely and easily. Limiting ourselves to the system authenticating consumables (we don't consider the consumable authenticating the system), two levels of protection can be considered:

**Presence Only Authentication**

This is where only the presence of an Authentication Chip is tested. The Authentication Chip can be reused in another consumable without being reprogrammed.

**Consumable Lifetime Authentication**

This is where not only is the presence of the Authentication Chip tested for, but also the Authentication chip 53 must only last the lifetime of the consumable. For the chip to be reused it must be completely erased and reprogrammed. The two levels of protection address different requirements. We are primarily concerned with Consumable Lifetime Authentication in order to prevent cloned versions of high volume consumables. In this case, each chip should hold secure state information about the consumable being authenticated. It should be noted that a Consumable Lifetime Authentication Chip could be used in any situation requiring a Presence Only Authentication Chip. The requirements for authentication, data storage integrity and manufacture should be considered separately. The following sections summarize requirements of each.

<u>Authentication</u>

The authentication requirements for both Presence Only Authentication and Consumable Lifetime Authentication are restricted to case of a system authenticating a consumable. For Presence Only Authentication, we must be assured that an Authentication Chip is physically present. For Consumable Lifetime Authentication we also need to be assured that state data actually came from the Authentication Chip, and that it has not been altered en route. These issues cannot be separated – data that has been altered has a new source, and if the source cannot be determined, the question of alteration cannot be settled. It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. The primary requirement therefore is to provide authentication by means that have withstood the scrutiny of experts. The authentication scheme used by the Authentication chip 53 should be resistant to defeat by logical means. Logical types of attack are extensive, and attempt to do one of three things:

Bypass the authentication process altogether

Obtain the secret key by force or deduction, so that any question can be answered

Find enough about the nature of the authenticating questions and answers in order to, without the key, give the right answer to each question.

<u>Data Storage Integrity</u>

Although Authentication protocols take care of ensuring data integrity in communicated messages, data storage integrity is also required. Two kinds of data must be stored within the Authentication Chip:

Authentication data, such as secret keys

Consumable state data, such as serial numbers, and media remaining etc.

The access requirements of these two data types differ greatly. The Authentication chip 53 therefore requires a storage/access control mechanism that allows for the integrity requirements of each type.

Authentication Data

Authentication data must remain confidential. It needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on must not be permitted to leave the chip. It must be resistant to being read from non-volatile memory. The authentication scheme is responsible for ensuring the key cannot be obtained by deduction, and the manufacturing process is responsible for ensuring that the key cannot be obtained by physical means. The size of the authentication data memory area must be large enough to hold the necessary keys and secret information as mandated by the authentication protocols.

Consumable State Data

Each Authentication chip 53 needs to be able to also store 256 bits (32 bytes) of consumable state data. Consumable state data can be divided into the following types. Depending on the application, there will be different numbers of each of these types of data items. A maximum number of 32 bits for a single data item is to be considered.

Read Only

ReadWrite

Decrement Only

**Read Only** data needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on should not be allowed to change. Examples of Read Only data items are consumable batch numbers and serial numbers.

**ReadWrite** data is changeable state information, for example, the last time the particular consumable was used. ReadWrite data items can be read and written an unlimited number of times during the lifetime of the consumable. They can be used to store any state information about the consumable. The only requirement for this data is that it needs to be kept in non-volatile memory. Since an attacker can obtain access to a system (which can write to ReadWrite data), any attacker can potentially change data fields of this type. This data type should not be used for secret information, and must be considered insecure.

**Decrement Only** data is used to count down the availability of consumable resources. A photocopier's toner cartridge, for example, may store the amount of toner remaining as a Decrement Only data item. An ink cartridge for a color printer may store the amount of each ink color as a Decrement Only data item, requiring 3 (one for each of Cyan, Magenta, and Yellow), or even as many as 5 or 6 Decrement Only data items. The requirement for this kind of data item is that once programmed with an initial value at the manufacturing/programming stage, it can only reduce in value. Once it reaches the minimum value, it cannot decrement any further. The Decrement Only data item is only required by Consumable Lifetime Authentication.

Manufacture

The Authentication chip 53 ideally must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables. The Authentication chip 53 should use a standard manufacturing process, such as Flash. This is necessary to:

Allow a great range of manufacturing location options

Use well-defined and well-behaved technology

Reduce cost

Regardless of the authentication scheme used, the circuitry of the authentication part of the chip must be resistant to physical attack. Physical attack comes in four main ways, although the form of the attack can vary:

Bypassing the Authentication Chip altogether

Physical examination of chip while in operation (destructive and non-destructive)

Physical decomposition of chip

Physical alteration of chip

Ideally, the chip should be exportable from the U.S., so it should not be possible to use an Authentication chip 53 as a secure encryption device. This is low priority requirement since there are many companies in other countries able to manufacture the Authentication chips. In any case, the export restrictions from the U.S. may change.

Authentication

Existing solutions to the problem of authenticating consumables have typically relied on physical patents on packaging. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. Consequently a much higher level of protection is required. It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. Security systems such as Netscape's original proprietary system and the GSM Fraud Prevention Network used by cellular phones are examples where design secrecy caused the vulnerability of the security. Both security systems were broken by conventional means that would have been detected if the companies had followed an open design process. The solution is to provide authentication by means that have withstood the scrutiny of experts. A number of protocols that can be used for consumables authentication. We only use security methods that are publicly described, using known behaviors in this new way. For all protocols, the security of the scheme relies on a secret key, not a secret algorithm. All the protocols rely on a time-variant challenge (i.e. the challenge is different each time), where the response depends on the challenge and the secret. The challenge involves a random number so that any observer will not be able to gather useful information about a subsequent identification. Two protocols are presented for each of Presence Only Authentication and Consumable Lifetime Authentication. Although the protocols differ in the number of Authentication Chips required for the authentication process, in all cases the System authenticates the consumable. Certain protocols will work with either one or two chips, while other protocols only work with two chips. Whether one chip or two Authentication Chips are used the System is still responsible for making the authentication decision.

Single Chip Authentication

When only one Authentication chip 53 is used for the authentication protocol, a single chip (referred to as **ChipA**) is responsible for proving to a system (referred to as **System**) that it is authentic. At the start of the protocol, System is unsure of ChipA's authenticity. System undertakes a challenge-response protocol with ChipA, and thus determines ChipA's authenticity. In all protocols the authenticity of the consumable is directly based on the authenticity of the chip, i.e. if ChipA is considered authentic, then the consumable is considered authentic. The data flow can be seen in Fig. 167. In single chip authentication protocols, System can be software, hardware or a combination of both. It is important to note that System is considered insecure – it can be easily reverse engineered by an attacker, either by examining the ROM or by examining circuitry. System is not specially engineered to be secure in itself.

Double Chip Authentication

In other protocols, two Authentication Chips are required as shown in Fig. 168. A single chip (referred to as **ChipA**) is responsible for proving to a system (referred to as **System**) that it is authentic. As part of the authentication process, System makes use of a trusted Authentication Chip (referred to as **ChipT**). In double chip authentication protocols, System can be software, hardware or a combination of both. However ChipT must be a physical Authentication Chip. In some protocols ChipT and ChipA have the same internal structure, while in others ChipT and ChipA have different internal structures.

Presence Only Authentication (Insecure State Data)

For this level of consumable authentication we are only concerned about validating the presence of the Authentication chip 53. Although the Authentication Chip can contain state information, the transmission of that state information would not be considered secure. Two protocols are presented. Protocol 1 requires 2 Authentication Chips, while Protocol 2 can be implemented using either 1 or 2 Authentication Chips.

Protocol 1

Protocol 1 is a double chip protocol (two Authentication Chips are required). Each Authentication Chip contains the following values:

**K** Key for $F_K[X]$. Must be secret.

**R** Current random number. Does not have to be secret, but must be seeded with a different initial value for each chip instance. Changes with each invocation of the Random function.

Each Authentication Chip contains the following logical functions:

**Random[]** Returns R, and advances R to next in sequence.

**F[X]** Returns $F_K[X]$, the result of applying a one-way function F to X based upon the secret key K.

The protocol is as follows:

System requests Random[] from ChipT;

ChipT returns R to System;

System requests F[R] from both ChipT and ChipA;

ChipT returns $F_{KT}[R]$ to System;

ChipA returns $F_{KA}[R]$ to System;

System compares $F_{KT}[R]$ with $F_{KA}[R]$. If they are equal, then ChipA is considered valid. If not, then ChipA is considered invalid.

The data flow can be seen in Fig. 169. The System does not have to comprehend $F_K[R]$ messages. It must merely check that the responses from ChipA and ChipT are the same. The System therefore does not require the key. The security of Protocol 1 lies in two places:

The security of F[X]. Only Authentication chips contain the secret key, so anything that can produce an F[X] from an X that matches the F[X] generated by a trusted Authentication chip 53 (ChipT) must be authentic.

The domain of R generated by all Authentication chips must be large and non-deterministic. If the domain of R generated by all Authentication chips is small, then there is no need for a clone manufacturer to crack the key. Instead, the clone manufacturer could incorporate a ROM in their chip that had a record of all of the responses from a genuine chip to the codes sent by the system. The Random function does not strictly have to be in the Authentication Chip, since System can potentially generate the same random number sequence.

However it simplifies the design of System and ensures the security of the random number generator will be the same for all implementations that use the Authentication Chip, reducing possible error in system implementation.

Protocol 1 has several advantages:

K is not revealed during the authentication process

Given X, a clone chip cannot generate $F_K[X]$ without K or access to a real Authentication Chip.

System is easy to design, especially in low cost systems such as ink-jet printers, as no encryption or decryption is required by System itself.

A wide range of keyed one-way functions exists, including symmetric cryptography, random number sequences, and message authentication codes.

One-way functions require fewer gates and are easier to verify than asymmetric algorithms).

Secure key size for a keyed one-way function does not have to be as large as for an asymmetric (public key) algorithm. A minimum of 128 bits can provide appropriate security if F[X] is a symmetric cryptographic function.

However there are problems with this protocol:

It is susceptible to chosen text attack. An attacker can plug the chip into their own system, generate chosen Rs, and observe the output. In order to find the key, an attacker can also search for an R that will generate a specific F[M] since multiple Authentication chips can be tested in parallel.

Depending on the one-way function chosen, key generation can be complicated. The method of selecting a good key depends on the algorithm being used. Certain keys are weak for a given algorithm.

The choice of the keyed one-way functions itself is non-trivial. Some require licensing due to patent protection.

A man-in the middle could take action on a plaintext message M before passing it on to ChipA – it would be preferable if the man-in-the-middle did not see M until after ChipA had seen it. It would be even more preferable if a man-in-the-middle didn't see M at all.

If F is symmetric encryption, because of the key size needed for adequate security, the chips could not be exported from the USA since they could be used as strong encryption devices.

If Protocol 1 is implemented with F as an asymmetric encryption algorithm, there is no advantage over the symmetric case – the keys needs to be longer and the encryption algorithm is more expensive in silicon. Protocol 1 must be implemented with 2 Authentication Chips in order to keep the key secure. This means that each System requires an Authentication Chip and each consumable requires an Authentication Chip.

Protocol 2

In some cases, System may contain a large amount of processing power. Alternatively, for instances of systems that are manufactured in large quantities, integration of ChipT into System may be desirable. Use of an asymmetrical encryption algorithm allows the ChipT portion of System to be insecure. Protocol 2 therefore, uses asymmetric cryptography. For this protocol, each chip contains the following values:

**K** Key for $E_K[X]$ and $D_K[X]$. Must be secret in ChipA. Does not have to be secret in ChipT.

**R** Current random number. Does not have to be secret, but must be seeded with a different initial value for each chip instance. Changes with each invocation of the Random function.

The following functions are defined:

**E[X]** ChipT only. Returns $E_K[X]$ where E is asymmetric encrypt function E.

**D[X]** ChipA only. Returns $D_K[X]$ where D is asymmetric decrypt function D.

**Random[]** ChipT only. Returns R | $E_K[R]$, where R is random number based on seed S. Advances R to next in random number sequence.

The public key $K_T$ is in ChipT, while the secret key $K_A$ is in ChipA. Having $K_T$ in ChipT has the advantage that ChipT can be implemented in software or hardware (with the proviso that the seed for R is different for each chip or system). Protocol 2 therefore can be implemented as a Single Chip Protocol or as a Double Chip Protocol. The protocol for authentication is as follows:

System calls ChipT's Random function;

ChipT returns R | $E_{KT}[R]$ to System;

System calls ChipA's D function, passing in $E_{KT}[R]$;

ChipA returns R, obtained by $D_{KA}[E_{KT}[R]]$;

System compares R from ChipA to the original R generated by ChipT. If they are equal, then ChipA is considered valid. If not, ChipA is invalid.

The data flow can be seen in Fig. 170. Protocol 2 has the following advantages:

$K_A$ (the secret key) is not revealed during the authentication process

Given $E_{KT}[X]$, a clone chip cannot generate X without $K_A$ or access to a real ChipA.

Since $K_T \neq K_A$, ChipT can be implemented completely in software or in insecure hardware or as part of System. Only ChipA (in the consumable) is required to be a secure Authentication Chip.

If ChipT is a physical chip, System is easy to design.

There are a number of well-documented and cryptanalyzed asymmetric algorithms to chose from for implementation, including patent-free and license-free solutions.

However, Protocol 2 has a number of its own problems:

For satisfactory security, each key needs to be 2048 bits (compared to minimum 128 bits for symmetric cryptography in Protocol 1). The associated intermediate memory used by the encryption and decryption algorithms is correspondingly larger.

Key generation is non-trivial. Random numbers are not good keys.

If ChipT is implemented as a core, there may be difficulties in linking it into a given System ASIC.

If ChipT is implemented as software, not only is the implementation of System open to programming error and non-rigorous testing, but the integrity of the compiler and mathematics primitives must be rigorously checked for each implementation of System. This is more complicated and costly than simply using a well-tested chip.

Although many symmetric algorithms are specifically strengthened to be resistant to differential cryptanalysis (which is based on chosen text attacks), the private key $K_A$ is susceptible to a chosen text attack

If ChipA and ChipT are instances of the same Authentication Chip, each chip must contain both asymmetric encrypt and decrypt functionality. Consequently each chip is larger, more complex, and more expensive than the chip required for Protocol 1.

If the Authentication Chip is broken into 2 chips to save cost and reduce complexity of design/test, two chips still need to be manufactured, reducing the economies of scale. This is offset by the relative numbers of systems to consumables, but must still be taken into account.

Protocol 2 Authentication Chips could not be exported from the USA, since they would be considered strong encryption devices.

Even if the process of choosing a key for Protocol 2 was straightforward, Protocol 2 is impractical at the present time due to the high cost of silicon implementation (both key size and functional implementation). Therefore Protocol 1 is the protocol of choice for Presence Only Authentication.

Clone Consumable using Real Authentication Chip

Protocols 1 and 2 only check that ChipA is a real Authentication Chip. They do not check to see if the consumable itself is valid. The fundamental assumption for authentication is that if ChipA is valid, the consumable is valid. It is therefore possible for a clone manufacturer to insert a real Authentication Chip into a clone consumable. There are two cases to consider:

In cases where state data is not written to the Authentication Chip, the chip is completely reusable. Clone manufacturers could therefore recycle a valid consumable into a clone consumable. This may be made more difficult by melding the Authentication Chip into the consumable's physical packaging, but it would not stop refill operators.

In cases where state data is written to the Authentication Chip, the chip may be new, partially used up, or completely used up. However this does not stop a clone manufacturer from using the Piggyback attack, where the clone manufacturer builds a chip that has a real Authentication Chip as a piggyback. The Attacker's chip (ChipE) is therefore a man-in-the-middle. At power up, ChipE reads all the memory state values from the real Authentication chip 53 into its own memory. ChipE then examines requests from System, and takes different actions depending on the request. Authentication requests can be passed directly to the real Authentication chip 53, while read/write requests can be simulated by a memory that resembles real Authentication Chip behavior. In this way the Authentication chip 53 will always appear fresh at power-up. ChipE can do this because the data access is not authenticated.

In order to fool System into thinking its data accesses were successful, ChipE still requires a real Authentication Chip, and in the second case, a clone chip is required in addition to a real Authentication Chip. Consequently Protocols 1 and 2 can be useful in situations where it is not cost effective for a clone manufacturer to embed a real Authentication chip 53 into the consumable. If the consumable cannot be recycled or refilled easily, it may be protection enough to use Protocols 1 or 2. For a clone operation to be successful each clone consumable must include a valid Authentication Chip. The chips would have to be stolen en masse, or taken from old consumables. The quantity of these reclaimed chips (as well as the effort in reclaiming them) should not be enough to base a business on, so the added protection of secure data transfer (see Protocols 3 and 4) may not be useful.

Longevity of Key

A general problem of these two protocols is that once the authentication key is chosen, it cannot easily be changed. In some instances a key-compromise is not a problem, while for others a key compromise is disastrous. For example, in a car/car-key System/Consumable scenario, the customer has only one set of car/car-keys. Each car has a different authentication key. Consequently the loss of a car-key only compromises the individual car. If the owner considers this a problem, they must get a new lock on the car by replacing the System chip inside the car's electronics. The owner's keys must be reprogrammed/replaced to work with the new car System Authentication Chip. By contrast, a compromise of a key for a high volume consumable market (for example ink cartridges in printers) would allow a

clone ink cartridge manufacturer to make their own Authentication Chips. The only solution for existing systems is to update the System Authentication Chips, which is a costly and logistically difficult exercise. In any case, consumers' Systems already work - they have no incentive to hobble their existing equipment.

Consumable Lifetime Authentication

In this level of consumable authentication we are concerned with validating the existence of the Authentication Chip, as well as ensuring that the Authentication Chip lasts only as long as the consumable. In addition to validating that an Authentication Chip is present, writes and reads of the Authentication Chip's memory space must be authenticated as well. In this section we assume that the Authentication Chip's data storage integrity is secure – certain parts of memory are Read Only, others are Read/Write, while others are Decrement Only (see the chapter entitled **Data Storage Integrity** for more information). Two protocols are presented. Protocol 3 requires 2 Authentication Chips, while Protocol 4 can be implemented using either 1 or 2 Authentication Chips.

Protocol 3

This protocol is a double chip protocol (two Authentication Chips are required). For this protocol, each Authentication Chip contains the following values:

**$K_1$** Key for calculating $F_{K1}[X]$. Must be secret.

**$K_2$** Key for calculating $F_{K2}[X]$. Must be secret.

**R** Current random number. Does not have to be secret, but must be seeded with a different initial value for each chip instance. Changes with each successful authentication as defined by the Test function.

**M** Memory vector of Authentication chip 53. Part of this space should be different for each chip (does not have to be a random number).

Each Authentication Chip contains the following logical functions:

**F[X]** Internal function only. Returns $F_K[X]$, the result of applying a one-way function F to X based upon either key $K_1$ or key $K_2$

**Random[]** Returns $R \mid F_{K1}[R]$.

**Test[X, Y]** Returns 1and advances R if $F_{K2}[R \mid X] = Y$. Otherwise returns 0. The time taken to return 0 must be identical for all bad inputs.

**Read[X, Y]** Returns $M \mid F_{K2}[X \mid M]$ if $F_{K1}[X] = Y$. Otherwise returns 0. The time taken to return 0 must be identical for all bad inputs.

**Write[X]** Writes X over those parts of M that can legitimately be written over.

To authenticate ChipA and read ChipA's memory M:

System calls ChipT's Random function;

ChipT produces $R \mid F_K[R]$ and returns these to System;

System calls ChipA's Read function, passing in R, $F_K[R]$;

ChipA returns M and $F_K[R \mid M]$;

System calls ChipT's Test function, passing in M and $F_K[R \mid M]$;

System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

To authenticate a write of $M_{new}$ to ChipA's memory M:

System calls ChipA's Write function, passing in $M_{new}$;

The authentication procedure for a Read is carried out;

If ChipA is authentic and $M_{new} = M$, the write succeeded. Otherwise it failed.

The data flow for read authentication is shown in Fig. 171. The first thing to note about Protocol 3 is that $F_K[X]$ cannot be called directly. Instead $F_K[X]$ is called indirectly by Random, Test and Read:

Random[] calls $F_{K1}[X]$ X is not chosen by the caller. It is chosen by the Random function. An attacker must perform a brute force search using multiple calls to Random, Read, and Test to obtain a desired X, $F_{K1}[X]$ pair.

Test[X,Y] calls $F_{K2}[R \mid X]$ Does not return result directly, but compares the result to Y and then returns 1 or 0. Any attempt to deduce $K_2$ by calling Test multiple times trying different values of $F_{K2}[R \mid X]$ for a given X is reduced to a brute force search where R cannot even be chosen by the attacker.

Read[X, Y] calls $F_{K1}[X]$ X and $F_{K1}[X]$ must be supplied by caller, so the caller must already know the X, $F_{K1}[X]$ pair. Since the call returns 0 if

$Y \neq F_{K1}[X]$, a caller can use the Read function for a brute force attack on $K_1$.

Read[X, Y] calls $F_{K2}[X \mid M]$, X is supplied by caller, however X can only be those values already given out by the Random function (since X and Y are validated via $K_1$). Thus a chosen text attack must first collect pairs from Random (effectively a brute force attack). In addition, only part of M can be used in a chosen text attack since some of M is constant (read-only) and the decrement-only part of M can only be used once per consumable. In the next consumable the read-only part of M will be different.

Having $F_K[X]$ being called indirectly prevents chosen text attacks on the Authentication Chip. Since an attacker can only obtain a chosen R, $F_{K1}[R]$ pair by calling Random, Read, and Test multiple times until the desired R appears, a brute force attack on $K_1$ is required in order to perform a limited chosen text attack on $K_2$. Any attempt at a chosen text attack on $K_2$ would be limited since the text cannot be completely chosen: parts of M are read-only, yet different for each Authentication Chip. The second thing to note is that two keys are used. Given the small size of M, two different keys $K_1$ and $K_2$ are used in order to ensure there is no correlation between F[R] and F[R|M]. $K_1$ is therefore used to help protect $K_2$ against differential attacks. It is not enough to use a single longer key since M is only 256 bits, and only part of M changes during the lifetime of the consumable. Otherwise it is potentially possible that an attacker via some as-yet undiscovered technique, could determine the effect of the limited changes in M to particular bit combinations in R and thus calculate $F_{K2}[X \mid M]$ based on $F_{K1}[X]$. As an added precaution, the Random and Test functions in ChipA should be disabled so that in order to generate R, $F_K[R]$ pairs, an attacker must use instances of ChipT, each of which is more expensive than ChipA (since a system must be obtained for each ChipT). Similarly, there should be a minimum delay between calls to Random, Read and Test so that an attacker cannot call these functions at high speed. Thus each chip can only give a specific number of X, $F_K[X]$ pairs away in a certain time period. The only specific timing requirement of Protocol 3 is that the return value of 0 (indicating a bad input) must be produced in the same amount of time regardless of where the error is in the input. Attackers can therefore not learn anything about what was bad about the input value. This is true for both RD and TST functions.

Another thing to note about Protocol 3 is that Reading data from ChipA also requires authentication of ChipA. The System can be sure that the contents of memory (M) is what ChipA claims it to be if $F_{K2}[R \mid M]$ is returned correctly. A clone chip may pretend that M is a certain value (for example it may pretend that the consumable is full), but it cannot return $F_{K2}[R \mid M]$ for any R passed in by System. Thus the effective signature $F_{K2}[R \mid M]$ assures System that not only

did an authentic ChipA send M, but also that M was not altered in between ChipA and System. Finally, the Write function as defined does not authenticate the Write. To authenticate a write, the System must perform a Read after each Write. There are some basic advantages with Protocol 3:

$K_1$ and $K_2$ are not revealed during the authentication process

Given X, a clone chip cannot generate $F_{K2}[X \mid M]$ without the key or access to a real Authentication Chip.

System is easy to design, especially in low cost systems such as ink-jet printers, as no encryption or decryption is required by System itself.

A wide range of key based one-way functions exists, including symmetric cryptography, random number sequences, and message authentication codes.

Keyed one-way functions require fewer gates and are easier to verify than asymmetric algorithms).

Secure key size for a keyed one-way function does not have to be as large as for an asymmetric (public key) algorithm. A minimum of 128 bits can provide appropriate security if F[X] is a symmetric cryptographic function.

Consequently, with Protocol 3, the only way to authenticate ChipA is to read the contents of ChipA's memory. The security of this protocol depends on the underlying $F_K[X]$ scheme and the domain of R over the set of all Systems. Although $F_K[X]$ can be any keyed one-way function, there is no advantage to implement it as asymmetric encryption. The keys need to be longer and the encryption algorithm is more expensive in silicon. This leads to a second protocol for use with asymmetric algorithms – Protocol 4. Protocol 3 must be implemented with 2 Authentication Chips in order to keep the keys secure. This means that each System requires an Authentication Chip and each consumable requires an Authentication Chip

<u>Protocol 4</u>

In some cases, System may contain a large amount of processing power. Alternatively, for instances of systems that are manufactured in large quantities, integration of ChipT into System may be desirable. Use of an asymmetrical encryption algorithm can allow the ChipT portion of System to be insecure. Protocol 4 therefore, uses asymmetric cryptography. For this protocol, each chip contains the following values:

**K** Key for $E_K[X]$ and $D_K[X]$. Must be secret in ChipA. Does not have to be secret in ChipT.

**R** Current random number. Does not have to be secret, but must be seeded with a different initial value for each chip instance. Changes with each successful authentication as defined by the Test function.

**M** Memory vector of Authentication chip 53. Part of this space should be different for each chip, (does not have to be a random number).

There is no point in verifying anything in the Read function, since anyone can encrypt using a public key. Consequently the following functions are defined:

**E[X]** Internal function only. Returns $E_K[X]$ where E is asymmetric encrypt function E.

**D[X]** Internal function only. Returns $D_K[X]$ where D is asymmetric decrypt function D.

**Random[]** ChipT only. Returns $E_K[R]$.

**Test[X, Y]** Returns 1 and advances R if $D_K[R \mid X] = Y$. Otherwise returns 0. The time taken to return 0 must be identical for all bad inputs.

**Read[X]** Returns $M \mid E_K[R \mid M]$ where $R = D_K[X]$ (does not test input).

**Write[X]** Writes X over those parts of M that can legitimately be written over.

The public key $K_T$ is in ChipT, while the secret key $K_A$ is in ChipA. Having $K_T$ in ChipT has the advantage that ChipT can be implemented in software or hardware (with the proviso that R is seeded with a different random number for each system). To authenticate ChipA and read ChipA's memory M:

System calls ChipT's Random function;

ChipT produces ad returns $E_{KT}[R]$ to System;

System calls ChipA's Read function, passing in $E_{KT}[R]$;

ChipA returns $M \mid E_{KA}[R \mid M]$, first obtaining R by $D_{KA}[E_{KT}[R]]$;

System calls ChipT's Test function, passing in M and $E_{KA}[R \mid M]$;

ChipT calculates $D_{KT}[E_{KA}[R \mid M]]$ and compares it to $R \mid M$.

System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

To authenticate a write of $M_{new}$ to ChipA's memory M:

System calls ChipA's Write function, passing in $M_{new}$;

The authentication procedure for a Read is carried out;

If ChipA is authentic and $M_{new} = M$, the write succeeded. Otherwise it failed.

The data flow for read authentication is shown in Fig. 172. Only a valid ChipA would know the value of R, since R is not passed into the Authenticate function (it is passed in as an encrypted value). R must be obtained by decrypting E[R], which can only be done using the secret key $K_A$. Once obtained, R must be appended to M and then the result re-encoded. ChipT can then verify that the decoded form of $E_{KA}[R \mid M] = R \mid M$ and hence ChipA is valid. Since $K_T \neq K_A$ , $E_{KT}[R] \neq E_{KA}[R]$. Protocol 4 has the following advantages:

$K_A$ (the secret key) is not revealed during the authentication process

Given $E_{KT}[X]$, a clone chip cannot generate X without $K_A$ or access to a real ChipA.

Since $K_T \neq K_A$, ChipT can be implemented completely in software or in insecure hardware or as part of System. Only ChipA is required to be a secure Authentication Chip.

Since ChipT and ChipA contain different keys, intense testing of ChipT will reveal nothing about $K_A$.

If ChipT is a physical chip, System is easy to design.

There are a number of well-documented and cryptanalyzed asymmetric algorithms to chose from for implementation, including patent-free and license-free solutions.

Even if System could be rewired so that ChipA requests were directed to ChipT, ChipT could never answer for ChipA since $K_T \neq K_A$. The attack would have to be directed at the System ROM itself to bypass the Authentication protocol.

However, Protocol 4 has a number of disadvantages:

All Authentication Chips need to contain both asymmetric encrypt and decrypt functionality. Consequently each chip is larger, more complex, and more expensive than the chip required for Protocol 3.

For satisfactory security, each key needs to be 2048 bits (compared to a minimum of 128 bits for symmetric cryptography in Protocol 1). The associated intermediate memory used by the encryption and decryption algorithms is correspondingly larger.

Key generation is non-trivial. Random numbers are not good keys.

If ChipT is implemented as a core, there may be difficulties in linking it into a given System ASIC.

If ChipT is implemented as software, not only is the implementation of System open to programming error and non-rigorous testing, but the integrity of the compiler and mathematics primitives must be rigorously checked for each implementation of System. This is more complicated and costly than simply using a well-tested chip.

Although many symmetric algorithms are specifically strengthened to be resistant to differential cryptanalysis (which is based on chosen text attacks), the private key $K_A$ is susceptible to a chosen text attack

Protocol 4 Authentication Chips could not be exported from the USA, since they would be considered strong encryption devices.

As with Protocol 3, the only specific timing requirement of Protocol 4 is that the return value of 0 (indicating a bad input) must be produced in the same amount of time regardless of where the error is in the input. Attackers can therefore not learn anything about what was bad about the input value. This is true for both RD and TST functions.

Variation on call to TST

If there are two Authentication Chips used, it is theoretically possible for a clone manufacturer to replace the System Authentication Chip with one that returns 1 (success) for each call to TST. The System can test for this by calling TST a number of times – N times with a wrong hash value, and expect the result to be 0. The final time that TST is called, the true returned value from ChipA is passed, and the return value is trusted. The question then arises of how many times to call TST. The number of calls must be random, so that a clone chip manufacturer cannot know the number ahead of time. If System has a clock, bits from the clock can be used to determine how many false calls to TST should be made. Otherwise the returned value from ChipA can be used. In the latter case, an attacker could still rewire the System to permit a clone ChipT to view the returned value from ChipA, and thus know which hash value is the correct one. The worst case of course, is that the System can be completely replaced by a clone System that does not require authenticated consumables – this is the limit case of rewiring and changing the System. For this reason, the variation on calls to TST is optional, depending on the System, the Consumable, and how likely modifications are to be made. Adding such logic to System (for example in the case of a small desktop printer) may be considered not worthwhile, as the System is made more complicated. By contrast, adding such logic to a camera may be considered worthwhile.

Clone Consumable using Real Authentication Chip

It is important to decrement the amount of consumable remaining before use that consumable portion. If the consumable is used first, a clone consumable could fake a loss of contact during a write to the special known address and then appear as a fresh new consumable. It is important to note that this attack still requires a real Authentication Chip in each consumable.

Longevity of Key

A general problem of these two protocols is that once the authentication keys are chosen, it cannot easily be changed. In some instances a key-compromise is not a problem, while for others a key compromise is disastrous.

Choosing a protocol

Even if the choice of keys for Protocols 2 and 4 was straightforward, both protocols are impractical at the present time due to the high cost of silicon implementation (both due to key size and functional implementation). Therefore Protocols 1 and 3 are the two protocols of choice. However, Protocols 1 and 3 contain much of the same components:

both require read and write access;

both require implementation of a keyed one-way function; and

both require random number generation functionality.

Protocol 3 requires an additional key ($K_2$), as well as some minimal state machine changes:

a state machine alteration to enable $F_{K1}[X]$ to be called during Random;

a Test function which calls $F_{K2}[X]$

a state machine alteration to the Read function to call $F_{K1}[X]$ and $F_{K2}[X]$

Protocol 3 only requires minimal changes over Protocol 1. It is more secure and can be used in all places where Presence Only Authentication is required (Protocol 1). It is therefore the protocol of choice. Given that Protocols 1 and 3 both make use of keyed one-way functions, the choice of one-way function is examined in more detail here. The following table outlines the attributes of the applicable choices. The attributes are worded so that the attribute is seen as an advantage.

| | Triple DES | Blowfish | RC5 | IDEA | Random Sequences | HMAC-MD5 | HMAC-SHA1 | HMAC-RIPEMD160 |
|---|---|---|---|---|---|---|---|---|
| Free of patents | | | | | | | | |
| Random key generation | | | | | | | | |
| Can be exported from the USA | | | | | | | | |
| Fast | | | | | | | | |
| Preferred Key Size (bits) for use in this application | 168 | 128 | 128 | 128 | 512 | 128 | 160 | 160 |
| Block size (bits) | 64 | 64 | 64 | 64 | 256 | 512 | 512 | 512 |
| Cryptanalysis Attack-Free (apart from weak keys) | | | | | | | | |
| Output size given input size N | ≥N | ≥N | ≥N | ≥N | 128 | 128 | 160 | 160 |
| Low storage requirements | | | | | | | | |
| Low silicon complexity | | | | | | | | |
| NSA designed | | | | | | | | |

An examination of the table shows that the choice is effectively between the 3 HMAC constructs and the Random Sequence. The problem of key size and key generation eliminates the Random Sequence. Given that a number of attacks have already been carried out on MD5 and since the hash result is only 128 bits, HMAC-MD5 is also eliminated. The choice is therefore between HMAC-SHA1 and HMAC-RIPEMD160. RIPEMD-160 is relatively new, and has not been as extensively cryptanalyzed as SHA1. However, SHA-1 was designed by the NSA, so this may be

seen by some as a negative attribute.

Given that there is not much between the two, SHA-1 will be used for the HMAC construct.

Choosing A Random Number Generator

Each of the protocols described (1-4) requires a random number generator. The generator must be "good" in the sense that the random numbers generated over the life of all Systems cannot be predicted. If the random numbers were the same for each System, an attacker could easily record the correct responses from a real Authentication Chip, and place the responses into a ROM lookup for a clone chip. With such an attack there is no need to obtain $K_1$ or $K_2$. Therefore the random numbers from each System must be different enough to be unpredictable, or non-deterministic. As such, the initial value for R (the random seed) should be programmed with a physically generated random number gathered from a physically random phenomenon, one where there is no information about whether a particular bit will be 1 or 0. The seed for R must NOT be generated with a computer-run random number generator. Otherwise the generator algorithm and seed may be compromised enabling an attacker to generate and therefore know the set of all R values in all Systems.

Having a different R seed in each Authentication Chip means that the first R will be both random and unpredictable across all chips. The question therefore arises of how to generate subsequent R values in each chip.

The base case is not to change R at all. Consequently R and $F_{K1}[R]$ will be the same for each call to Random[]. If they are the same, then $F_{K1}[R]$ can be a constant rather than calculated. An attacker could then use a single valid Authentication Chip to generate a valid lookup table, and then use that lookup table in a clone chip programmed especially for that System. A constant R is not secure.

The simplest conceptual method of changing R is to increment it by 1. Since R is random to begin with, the values across differing systems are still likely to be random. However given an initial R, all subsequent R values can be determined directly (there is no need to iterate 10,000 times – R will take on values from $R_0$ to $R_0$+ 10000). An incrementing R is immune to the earlier attack on a constant R. Since R is always different, there is no way to construct a lookup table for the particular System without wasting as many real Authentication Chips as the clone chip will replace.

Rather than increment using an adder, another way of changing R is to implement it as an LFSR (Linear Feedback Shift Register). This has the advantage of less silicon than an adder, but the advantage of an attacker not being able to directly determine the range of R for a particular System, since an LFSR value-domain is determined by sequential access. To determine which values an given initial R will generate, an attacker must iterate through the possibilities and enumerate them. The advantages of a changing R are also evident in the LFSR solution. Since R is always different, there is no way to construct a lookup table for the particular System without using-up as many real Authentication Chips as the clone chip will replace (and only for that System). There is therefore no advantage in having a more complex function to change R. Regardless of the function, it will always be possible for an attacker to iterate through the lifetime set of values in a simulation. The primary security lies in the initial randomness of R. Using an LFSR to change R (apart from using less silicon than an adder) simply has the advantage of not being restricted to a consecutive numeric range (i.e. knowing R, $R_N$ cannot be directly calculated; an attacker must iterate through the LFSR N times).

The Random number generator within the Authentication Chip is therefore an LFSR with 160 bits. Tap selection of the 160 bits for a maximal-period LFSR (i.e. the LFSR will cycle through all $2^{160}-1$ states, 0 is not a valid state) yields bits 159, 4, 2, and 1, as shown in Fig. 173. The LFSR is sparse, in that not many bits are used for feedback (only 4 out of

160 bits are used). This is a problem for cryptographic applications, but not for this application of non-sequential number generation. The 160-bit seed value for R can be any random number except 0, since an LFSR filled with 0s will produce a never-ending stream of 0s. Since the LFSR described is a maximal period LFSR, all 160 bits can be used directly as R. There is no need to construct a number sequentially from output bits of $b_0$. After each successful call to TST, the random number (R) must be advanced by XORing bits 1, 2, 4, and 159, and shifting the result into the high order bit. The new R and corresponding $F_{K1}[R]$ can be retrieved on the next call to Random.

Holding out Against Logical Attacks

Protocol 3 is the authentication scheme used by the Authentication Chip. As such, it should be resistant to defeat by logical means. While the effect of various types of attacks on Protocol 3 have been mentioned in discussion, this section details each type of attack in turn with reference to Protocol 3.

Brute Force attack

A Brute Force attack is guaranteed to break Protocol 3. However the length of the key means that the time for an attacker to perform a brute force attack is too long to be worth the effort. An attacker only needs to break $K_2$ to build a clone Authentication Chip. $K_1$ is merely present to strengthen $K_2$ against other forms of attack. A Brute Force Attack on $K_2$ must therefore break a 160-bit key. An attack against $K_2$ requires a maximum of $2^{160}$ attempts, with a 50% chance of finding the key after only $2^{159}$ attempts. Assuming an array of a trillion processors, each running one million tests per second, $2^{159}$ ($7.3 \times 10^{47}$) tests takes $2.3 \times 10^{23}$ years, which is longer than the lifetime of the universe. There are only 100 million personal computers in the world. Even if these were all connected in an attack (e.g. via the Internet), this number is still 10,000 times smaller than the trillion-processor attack described. Further, if the manufacture of one trillion processors becomes a possibility in the age of nanocomputers, the time taken to obtain the key is longer than the lifetime of the universe.

Guessing the key attack

It is theoretically possible that an attacker can simply "guess the key". In fact, given enough time, and trying every possible number, an attacker will obtain the key. This is identical to the Brute Force attack described above, where $2^{159}$ attempts must be made before a 50% chance of success is obtained. The chances of someone simply guessing the key on the first try is $2^{160}$. For comparison, the chance of someone winning the top prize in a U.S. state lottery and being killed by lightning in the same day is only 1 in $2^{61}$. The chance of someone guessing the Authentication Chip key on the first go is 1 in $2^{160}$, which is comparative to two people choosing exactly the same atoms from a choice of all the atoms in the Earth i.e. extremely unlikely.

Quantum Computer attack

To break $K_2$, a quantum computer containing 160 qubits embedded in an appropriate algorithm must be built. An attack against a 160-bit key is not feasible. An outside estimate of the possibility of quantum computers is that 50 qubits may be achievable within 50 years. Even using a 50 qubit quantum computer, $2^{110}$ tests are required to crack a 160 bit key. Assuming an array of 1 billion 50 qubit quantum computers, each able to try $2^{50}$ keys in 1 microsecond (beyond the current wildest estimates) finding the key would take an average of 18 billion years.

Cyphertext Only attack

An attacker can launch a Cyphertext Only attack on $K_1$ by calling monitoring calls to RND and RD, and on $K_2$ by monitoring calls to RD and TST. However, given that all these calls also reveal the plaintext as well as the hashed form of the plaintext, the attack would be transformed into a stronger form of attack – a Known Plaintext attack.

Known Plaintext attack

It is easy to connect a logic analyzer to the connection between the System and the Authentication Chip, and thereby monitor the flow of data. This flow of data results in known plaintext and the hashed form of the plaintext, which can therefore be used to launch a Known Plaintext attack against both $K_1$ and $K_2$. To launch an attack against $K_1$, multiple calls to RND and TST must be made (with the call to TST being successful, and therefore requiring a call to RD on a valid chip). This is straightforward, requiring the attacker to have both a System Authentication Chip and a Consumable Authentication Chip. For each $K_1$ X, $H_{K1}[X]$ pair revealed, a $K_2$ Y, $H_{K2}[Y]$ pair is also revealed. The attacker must collect these pairs for further analysis. The question arises of how many pairs must be collected for a meaningful attack to be launched with this data. An example of an attack that requires collection of data for statistical analysis is Differential Cryptanalysis. However, there are no known attacks against SHA-1 or HMAC-SHA1, so there is no use for the collected data at this time.

Chosen Plaintext attacks

Given that the cryptanalyst has the ability to modify subsequent chosen plaintexts based upon the results of previous experiments, $K_2$ is open to a partial form of the Adaptive Chosen Plaintext attack, which is certainly a stronger form of attack than a simple Chosen Plaintext attack. A chosen plaintext attack is not possible against $K_1$, since there is no way for a caller to modify R, which used as input to the RND function (the only function to provide the result of hashing with $K_1$). Clearing R also has the effect of clearing the keys, so is not useful, and the SSI command calls CLR before storing the new R-value.

Adaptive Chosen plaintext attacks

This kind of attack is not possible against $K_1$, since $K_1$ is not susceptible to chosen plaintext attacks. However, a partial form of this attack is possible against $K_2$, especially since both System and consumables are typically available to the attacker (the System may not be available to the attacker in some instances, such as a specific car). The HMAC construct provides security against all forms of chosen plaintext attacks. This is primarily because the HMAC construct has 2 secret input variables (the result of the original hash, and the secret key). Thus finding collisions in the hash function itself when the input variable is secret is even harder than finding collisions in the plain hash function. This is because the former requires direct access to SHA-1 (not permitted in Protocol 3) in order to generate pairs of input/output from SHA-1. The only values that can be collected by an attacker are HMAC[R] and HMAC[R | M]. These are not attacks against the SHA-1 hash function itself, and reduce the attack to a Differential Cryptanalysis attack, examining statistical differences between collected data. Given that there is no Differential Cryptanalysis attack known against SHA-1 or HMAC, Protocol 3 is resistant to the Adaptive Chosen Plaintext attacks.

Purposeful Error Attack

An attacker can only launch a Purposeful Error Attack on the TST and RD functions, since these are the only functions that validate input against the keys. With both the TST and RD functions, a 0 value is produced if an error is found in the input – no further information is given. In addition, the time taken to produce the 0 result is independent of the input, giving the attacker no information about which bit(s) were wrong. A Purposeful Error Attack is therefore fruitless.

Chaining attack

Any form of chaining attack assumes that the message to be hashed is over several blocks, or the input variables can somehow be set. The HMAC-SHA1 algorithm used by Protocol 3 only ever hashes a single 512-bit block at a time.

Consequently chaining attacks are not possible against Protocol 3.

Birthday attack

The strongest attack known against HMAC is the birthday attack, based on the frequency of collisions for the hash function. However this is totally impractical for minimally reasonable hash functions such as SHA-1. And the birthday attack is only possible when the attacker has control over the message that is signed. Protocol 3 uses hashing as a form of digital signature. The System sends a number that must be incorporated into the response from a valid Authentication Chip. Since the Authentication Chip must respond with H[R | M], but has no control over the input value R, the birthday attack is not possible. This is because the message has effectively already been generated and signed. An attacker must instead search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday). The clone chip must therefore attempt to find a new value $R_2$ such that the hash of $R_2$ and a chosen $M_2$ yields the same hash value as H[R | M]. However the System Authentication Chip does not reveal the correct hash value (the TST function only returns 1 or 0 depending on whether the hash value is correct). Therefore the only way of finding out the correct hash value (in order to find a collision) is to interrogate a real Authentication Chip. But to find the correct value means to update M, and since the decrement-only parts of M are one-way, and the read-only parts of M cannot be changed, a clone consumable would have to update a real consumable before attempting to find a collision. The alternative is a Brute Force attack search on the TST function to find a success (requiring each clone consumable to have access to a System consumable). A Brute Force Search, as described above, takes longer than the lifetime of the universe, in this case, per authentication. Due to the fact that a timely gathering of a hash value implies a real consumable must be decremented, there is no point for a clone consumable to launch this kind of attack.

Substitution with a complete lookup table

The random number seed in each System is 160 bits. The worst case situation for an Authentication Chip is that no state data is changed. Consequently there is a constant value returned as M. However a clone chip must still return $F_{K2}$[R | M], which is a 160 bit value. Assuming a 160-bit lookup of a 160-bit result, this requires $7.3 \times 10^{48}$ bytes, or $6.6 \times 10^{36}$ terabytes, certainly more space than is feasible for the near future. This of course does not even take into account the method of collecting the values for the ROM. A complete lookup table is therefore completely impossible.

Substitution with a sparse lookup table

A sparse lookup table is only feasible if the messages sent to the Authentication Chip are somehow predictable, rather than effectively random. The random number R is seeded with an unknown random number, gathered from a naturally random event. There is no possibility for a clone manufacturer to know what the possible range of R is for all Systems, since each bit has a 50% chance of being a 1 or a 0. Since the range of R in all systems is unknown, it is not possible to build a sparse lookup table that can be used in **all** systems. The general sparse lookup table is therefore not a possible attack. However, it is possible for a clone manufacturer to know what the range of R is for a given System. This can be accomplished by loading a LFSR with the current result from a call to a specific System Authentication Chip's RND function, and iterating some number of times into the future. If this is done, a special ROM can be built which will only contain the responses for that particular range of R, i.e. a ROM specifically for the consumables of that particular System. But the attacker still needs to place correct information in the ROM. The attacker will therefore need to find a valid Authentication Chip and call it for each of the values in R.

Suppose the clone Authentication Chip reports a full consumable, and then allows a single use before simulating loss

of connection and insertion of a new full consumable. The clone consumable would therefore need to contain responses for authentication of a full consumable and authentication of a partially used consumable. The worst case ROM contains entries for full and partially used consumables for R over the lifetime of System. However, a valid Authentication Chip must be used to generate the information, and be partially used in the process. If a given System only produces about n R-values, the sparse lookup-ROM required is 10n bytes multiplied by the number of different values for M. The time taken to build the ROM depends on the amount of time enforced between calls to RD.

After all this, the clone manufacturer must rely on the consumer returning for a refill, since the cost of building the ROM in the first place consumes a single consumable. The clone manufacturer's business in such a situation is consequently in the refills. The time and cost then, depends on the size of R and the number of different values for M that must be incorporated in the lookup. In addition, a custom clone consumable ROM must be built to match each and every System, and a different valid Authentication Chip must be used for each System (in order to provide the full and partially used data). The use of an Authentication Chip in a System must therefore be examined to determine whether or not this kind of attack is worthwhile for a clone manufacturer. As an example, of a camera system that has about 10,000 prints in its lifetime. Assume it has a single Decrement Only value (number of prints remaining), and a delay of 1 second between calls to RD. In such a system, the sparse table will take about 3 hours to build, and consumes 100K. Remember that the construction of the ROM requires the consumption of a valid Authentication Chip, so any money charged must be worth more than a single consumable and the clone consumable combined. Thus it is not cost effective to perform this function for a single consumable (unless the clone consumable somehow contained the equivalent of multiple authentic consumables). If a clone manufacturer is going to go to the trouble of building a custom ROM for each owner of a System, an easier approach would be to update System to completely ignore the Authentication Chip.

Consequently, this attack is possible as a per-System attack, and a decision must be made about the chance of this occurring for a given System/Consumable combination. The chance will depend on the cost of the consumable and Authentication Chips, the longevity of the consumable, the profit margin on the consumable, the time taken to generate the ROM, the size of the resultant ROM, and whether customers will come back to the clone manufacturer for refills that use the same clone chip etc.

Differential cryptanalysis

Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although other algorithms such as HMAC-SHA1 used in Protocol 3 have no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

Minimal-difference inputs, and their corresponding outputs

Minimal-difference outputs, and their corresponding inputs

To launch an attack of this nature, sets of input/output pairs must be collected. The collection from Protocol 3 can be via Known Plaintext, or from a Partially Adaptive Chosen Plaintext attack. Obviously the latter, being chosen, will be more useful. Hashing algorithms in general are designed to be resistant to differential analysis. SHA-1 in particular has been specifically strengthened, especially by the 80 word expansion so that minimal differences in input produce will still produce outputs that vary in a larger number of bit positions (compared to 128 bit hash functions). In addition, the information collected is not a direct SHA-1 input/output set, due to the nature of the HMAC algorithm. The HMAC algorithm hashes a known value with an unknown value (the key), and the result of this hash is then rehashed with a

separate unknown value. Since the attacker does not know the secret value, nor the result of the first hash, the inputs and outputs from SHA-1 are not known, making any differential attack extremely difficult. The following is a more detailed discussion of minimally different inputs and outputs from the Authentication Chip.

**Minimal Difference Inputs**

This is where an attacker takes a set of X, $F_K[X]$ values where the X values are minimally different, and examines the statistical differences between the outputs $F_K[X]$. The attack relies on X values that only differ by a minimal number of bits. The question then arises as to how to obtain minimally different X values in order to compare the $F_K[X]$ values.

**$K_1$:** With $K_1$, the attacker needs to statistically examine minimally different X, $F_{K1}[X]$ pairs. However the attacker cannot choose any X value and obtain a related $F_{K1}[X]$ value. Since X, $F_{K1}[X]$ pairs can only be generated by calling the RND function on a System Authentication Chip, the attacker must call RND multiple times, recording each observed pair in a table. A search must then be made through the observed values for enough minimally different X values to undertake a statistical analysis of the $F_{K1}[X]$ values.

**$K_2$:** With $K_2$, the attacker needs to statistically examine minimally different X, $F_{K2}[X]$ pairs. The only way of generating X, $F_{K2}[X]$ pairs is via the RD function, which produces $F_{K2}[X]$ for a given Y, $F_{K1}[Y]$ pair, where $X = Y \mid M$. This means that Y and the changeable part of M can be chosen to a limited extent by an attacker. The amount of choice must therefore be limited as much as possible.

The first way of limiting an attacker's choice is to limit Y, since RD requires an input of the format Y, $F_{K1}[Y]$. Although a valid pair can be readily obtained from the RND function, it is a pair of RND's choosing. An attacker can only provide their own Y if they have obtained the appropriate pair from RND, or if they know $K_1$. Obtaining the appropriate pair from RND requires a Brute Force search. Knowing $K_1$ is only logically possible by performing cryptanalysis on pairs obtained from the RND function – effectively a known text attack. Although RND can only be called so many times per second, $K_1$ is common across System chips. Therefore known pairs can be generated in parallel.

The second way to limit an attacker's choice is to limit M, or at least the attacker's ability to choose M. The limiting of M is done by making some parts of M Read Only, yet different for each Authentication Chip, and other parts of M Decrement Only. The Read Only parts of M should ideally be different for each Authentication Chip, so could be information such as serial numbers, batch numbers, or random numbers. The Decrement Only parts of M mean that for an attacker to try a different M, they can only decrement those parts of M so many times – after the Decrement Only parts of M have been reduced to 0 those parts cannot be changed again. Obtaining a new Authentication chip 53 provides a new M, but the Read Only portions will be different from the previous Authentication Chip's Read Only portions, thus reducing an attacker's ability to choose M even further. Consequently an attacker can only gain a limited number of chances at choosing values for Y and M.

**Minimal Difference Outputs**

This is where an attacker takes a set of X, $F_K[X]$ values where the $F_K[X]$ values are minimally different, and examines the statistical differences between the X values. The attack relies on $F_K[X]$ values that only differ by a minimal number of bits. For both $K_1$ and $K_2$, there is no way for an attacker to generate an X value for a given $F_K[X]$. To do so would violate the fact that F is a one-way function. Consequently the only way for an attacker to mount an attack of this nature is to record all observed X, $F_K[X]$ pairs in a table. A search must then be made through the observed values for enough minimally different $F_K[X]$ values to undertake a statistical analysis of the X values. Given that this requires

more work than a minimally different input attack (which is extremely limited due to the restriction on M and the choice of R), this attack is not fruitful.

Message substitution attacks

In order for this kind of attack to be carried out, a clone consumable must contain a real Authentication chip 53, but one that is effectively reusable since it never gets decremented. The clone Authentication Chip would intercept messages, and substitute its own. However this attack does not give success to the attacker. A clone Authentication Chip may choose not to pass on a WR command to the real Authentication Chip. However the subsequent RD command must return the correct response (as if the WR had succeeded). To return the correct response, the hash value must be known for the specific R and M. As described in the Birthday Attack section, an attacker can only determine the hash value by actually updating M in a real Chip, which the attacker does not want to do. Even changing the R sent by System does not help since the System Authentication Chip must match the R during a subsequent TST. A Message substitution attack would therefore be unsuccessful. This is only true if System updates the amount of consumable remaining before it is used.

Reverse engineering the key generator

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the Netscape security program was initially broken.

Bypassing authentication altogether

Protocol 3 requires the System to update the consumable state data before the consumable is used, and follow every write by a read (to authenticate the write). Thus each use of the consumable requires an authentication. If the System adheres to these two simple rules, a clone manufacturer will have to simulate authentication via a method above (such as sparse ROM lookup).

Reuse of Authentication Chips

As described above, Protocol 3 requires the System to update the consumable state data before the consumable is used, and follow every write by a read (to authenticate the write). Thus each use of the consumable requires an authentication. If a consumable has been used up, then its Authentication Chip will have had the appropriate state-data values decremented to 0. The chip can therefore not be used in another consumable. Note that this only holds true for Authentication Chips that hold Decrement-Only data items. If there is no state data decremented with each usage, there is nothing stopping the reuse of the chip. This is the basic difference between Presence-Only Authentication and Consumable Lifetime Authentication. Protocol 3 allows both. The bottom line is that if a consumable has Decrement Only data items that are used by the System, the Authentication Chip cannot be reused without being completely reprogrammed by a valid Programming Station that has knowledge of the secret key.

Management decision to omit authentication to save costs

Although not strictly an external attack, a decision to omit authentication in future Systems in order to save costs will have widely varying effects on different markets. In the case of high volume consumables, it is essential to remember that it is very difficult to introduce authentication after the market has started, as systems requiring authenticated consumables will not work with older consumables still in circulation. Likewise, it is impractical to discontinue authentication at any stage, as older Systems will not work with the new, unauthenticated, consumables. In he second case, older Systems can be individually altered by replacing the System Authentication Chip by a simple chip that has

the same programming interface, but whose TST function always succeeds. Of course the System may be programmed to test for an always-succeeding TST function, and shut down. In the case of a specialized pairing, such as a car/car-keys, or door/door-key, or some other similar situation, the omission of authentication in future systems is trivial and non-repercussive. This is because the consumer is sold the entire set of System and Consumable Authentication Chips at the one time.

Garrote/bribe attack

This form of attack is only successful in one of two circumstances:

$K_1$, $K_2$, and R are already recorded by the chip-programmer, or

the attacker can coerce future values of $K_1$, $K_2$, and R to be recorded.

If humans or computer systems external to the Programming Station do not know the keys, there is no amount of force or bribery that can reveal them. The level of security against this kind of attack is ultimately a decision for the System/Consumable owner, to be made according to the desired level of service. For example, a car company may wish to keep a record of all keys manufactured, so that a person can request a new key to be made for their car. However this allows the potential compromise of the entire key database, allowing an attacker to make keys for any of the manufacturer's existing cars. It does not allow an attacker to make keys for any new cars. Of course, the key database itself may also be encrypted with a further key that requires a certain number of people to combine their key portions together for access. If no record is kept of which key is used in a particular car, there is no way to make additional keys should one become lost. Thus an owner will have to replace his car's Authentication Chip and all his car-keys. This is not necessarily a bad situation. By contrast, in a consumable such as a printer ink cartridge, the one key combination is used for all Systems and all consumables. Certainly if no backup of the keys is kept, there is no human with knowledge of the key, and therefore no attack is possible. However, a no-backup situation is not desirable for a consumable such as ink cartridges, since if the key is lost no more consumables can be made. The manufacturer should therefore keep a backup of the key information in several parts, where a certain number of people must together combine their portions to reveal the full key information. This may be required if case the chip programming station needs to be reloaded. In any case, none of these attacks are against Protocol 3 itself, since no humans are involved in the authentication process. Instead, it is an attack against the programming stage of the chips.

HMAC-SHA1

The mechanism for authentication is the HMAC-SHA1 algorithm, acting on one of:

HMAC-SHA1 (R, $K_1$), or

HMAC-SHA1 (R | M, $K_2$)

We will now examine the HMAC-SHA1 algorithm in greater detail than covered so far, and describes an optimization of the algorithm that requires fewer memory resources than the original definition.

HMAC

The HMAC algorithm proceeds, given the following definitions:

H = the hash function (e.g. MD5 or SHA-1)

n = number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5)

M = the data to which the MAC function is to be applied

K = the secret key shared by the two parties

ipad= 0x36 repeated 64 times

opad = 0x5C repeated 64 times

The HMAC algorithm is as follows:

Extend K to 64 bytes by appending 0x00 bytes to the end of K

XOR the 64 byte string created in (1) with ipad

Append data stream M to the 64 byte string created in (2)

Apply H to the stream generated in (3)

XOR the 64 byte string created in (1) with opad

Append the H result from (4) to the 64 byte string resulting from (5)

Apply H to the output of (6) and output the result

Thus:

$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad) \mid M]]$

HMAC-SHA1 algorithm is simply HMAC with H = SHA-1.

SHA-1

The SHA1 hashing algorithm is defined in the algorithm as summarized here.

Nine 32-bit constants are defined. There are 5 constants used to initialize the chaining variables, and there are 4 additive constants.

| | **Initial Chaining Values** | | **Additive Constants** |
|---|---|---|---|
| $h_1$ | 0x67452301 | $y_1$ | 0x5A827999 |
| $h_2$ | 0xEFCDAB89 | $y_2$ | 0x6ED9EBA1 |
| $h_3$ | 0x98BADCFE | $y_3$ | 0x8F1BBCDC |
| $h_4$ | 0x10325476 | $y_4$ | 0xCA62C1D6 |
| $h_5$ | 0xC3D2E1F0 | | |

Non-optimized SHA-1 requires a total of 2912 bits of data storage:

Five 32-bit chaining variables are defined: $H_1$, $H_2$, $H_3$, $H_4$ and $H_5$.

Five 32-bit working variables are defined: A, B, C, D, and E.

One 32-bit temporary variable is defined: t.

Eighty 32-bit temporary registers are defined: $X_{0-79}$.

The following functions are defined for SHA-1:

| Symbolic Nomenclature | Description |
|---|---|
| + | Addition modulo $2^{32}$ |
| X Y | Result of rotating X left through Y bit positions |
| f(X, Y, Z) | $(X \wedge Y) \vee (\sim X \wedge Z)$ |
| g(X, Y, Z) | $(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$ |
| h(X, Y, Z) | $X \oplus Y \oplus Z$ |

The hashing algorithm consists of firstly padding the input message to be a multiple of 512 bits and initializing the chaining variables $H_{1-5}$ with $h_{1-5}$. The padded message is then processed in 512-bit chunks, with the output hash value being the final 160-bit value given by the concatenation of the chaining variables: $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$. The steps of the SHA-1 algorithm are now examined in greater detail.

Step 1. Preprocessing

The first step of SHA-1 is to pad the input message to be a multiple of 512 bits as follows and to initialize the chaining variables.

| Steps to follow to preprocess the input message | |
|---|---|
| Pad the input message | Append a 1 bit to the message |
| | Append 0 bits such that the length of the padded message is 64-bits short of a multiple of 512 bits. |
| | Append a 64-bit value containing the length in bits of the original input message. Store the length as most significant bit through to least significant bit. |
| Initialize the chaining variables | $H_1 \leftarrow h_1, H_2 \leftarrow h_2, H_3 \leftarrow h_3, H_4 \leftarrow h_4, H_5 \leftarrow h_5$ |

Step 2. Processing

The padded input message can now be processed. We process the message in 512-bit blocks. Each 512-bit block is in the form of 16 x 32-bit words, referred to as $InputWord_{0-15}$.

| **Steps to follow for each 512 bit block ($InputWord_{0\text{-}15}$)** | |
|---|---|
| Copy the 512 input bits into $X_{0\text{-}15}$ | For j=0 to 15<br>$X_j$ = $InputWord_j$ |
| Expand $X_{0\text{-}15}$ into $X_{16\text{-}79}$ | For j=16 to 79<br>$X_j \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16})\quad 1)$ |
| Initialize working variables | $A \leftarrow H_1, B \leftarrow H_2, C \leftarrow H_3, D \leftarrow H_4, E \leftarrow H_5$ |
| Round 1 | For j=0 to 19<br>$t \leftarrow ((A\quad 5) + f(B, C, D) + E + X_j + y_1)$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\quad 30), B \leftarrow A, A \leftarrow t$ |
| Round 2 | For j = 20 to 39<br>$t \leftarrow ((A\quad 5) + h(B, C, D) + E + X_j + y_2)$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\quad 30), B \leftarrow A, A \leftarrow t$ |
| Round 3 | For j = 40 to 59<br>$t \leftarrow ((A\quad 5) + g(B, C, D) + E + X_j + y_3)$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\quad 30), B \leftarrow A, A \leftarrow t$ |
| Round 4 | For j = 60 to 79<br>$t \leftarrow ((A\quad 5) + h(B, C, D) + E + X_j + y_4)$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\quad 30), B \leftarrow A, A \leftarrow t$ |
| Update chaining variables | $H_1 \leftarrow H_1 + A, H_2 \leftarrow H_2 + B,$<br>$H_3 \leftarrow H_3 + C, H_4 \leftarrow H_4 + D,$<br>$H_5 \leftarrow H_5 + E$ |

Step 3. Completion

After all the 512-bit blocks of the padded input message have been processed, the output hash value is the final 160-bit value given by: $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$.

Optimization for Hardware Implementation

The SHA-1 Step 2 procedure is not optimized for hardware. In particular, the 80 temporary 32-bit registers use up valuable silicon on a hardware implementation. This section describes an optimization to the SHA-1 algorithm that only uses 16 temporary registers. The reduction in silicon is from 2560 bits down to 512 bits, a saving of over 2000 bits. It may not be important in some applications, but in the Authentication Chip storage space must be reduced where possible. The optimization is based on the fact that although the original 16-word message block is expanded into an 80-word message block, the 80 words are not updated during the algorithm. In addition, the words rely on the previous 16 words only, and hence the expanded words can be calculated on-the-fly during processing, as long as we keep 16 words for the backward references. We require rotating counters to keep track of which register we are up to using, but the effect is to save a large amount of storage. Rather than index X by a single value j, we use a 5 bit counter to count

through the iterations. This can be achieved by initializing a 5-bit register with either 16 or 20, and decrementing it until it reaches 0. In order to update the 16 temporary variables as if they were 80, we require 4 indexes, each a 4-bit register. All 4 indexes increment (with wraparound) during the course of the algorithm.

| **Steps to follow for each 512 bit block (InputWord$_{0\text{-}15}$)** | |
|---|---|
| Initialize working variables | $A \leftarrow H_1, B \leftarrow H_2, C \leftarrow H_3, D \leftarrow H_4, E \leftarrow H_5$<br>$N_1 \leftarrow 13, N_2 \leftarrow 8, N_3 \leftarrow 2, N_4 \leftarrow 0$ |
| Round 0<br>Copy the 512 input bits into $X_{0\text{-}15}$ | Do 16 times:<br>$X_{N4} = \text{InputWord}_{N4}$<br>$[\ N_1,\ \ N_2,\ \ N_3]_{\text{optional}}\ \ N_4$ |
| Round 1A | Do 16 times:<br>$t \leftarrow ((A\ \ 5) + f(B, C, D) + E + X_{N4} + y_1)$<br>$[\ N_1,\ \ N_2,\ \ N_3]_{\text{optional}}\ \ N_4$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\ \ 30), B \leftarrow A, A \leftarrow t$ |
| Round 1B | Do 4 times:<br>$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4})\ \ \ 1)$<br>$t \leftarrow ((A\ \ 5) + f(B, C, D) + E + X_{N4} + y_1)$<br>$N_1,\ \ N_2,\ \ N_3,\ \ N_4$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\ \ 30), B \leftarrow A, A \leftarrow t$ |
| Round 2 | Do 20 times:<br>$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4})\ \ \ 1)$<br>$t \leftarrow ((A\ \ 5) + h(B, C, D) + E + X_{N4} + y_2)$<br>$N_1,\ \ N_2,\ \ N_3,\ \ N_4$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\ \ 30), B \leftarrow A, A \leftarrow t$ |
| Round 3 | Do 20 times:<br>$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4})\ \ \ 1)$<br>$t \leftarrow ((A\ \ 5) + g(B, C, D) + E + X_{N4} + y_3)$<br>$N_1,\ \ N_2,\ \ N_3,\ \ N_4$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\ \ 30), B \leftarrow A, A \leftarrow t$ |
| Round 4 | Do 20 times:<br>$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4})\ \ \ 1)$<br>$t \leftarrow ((A\ \ 5) + h(B, C, D) + E + X_{N4} + y_4)$<br>$N_1,\ \ N_2,\ \ N_3,\ \ N_4$<br>$E \leftarrow D, D \leftarrow C, C \leftarrow (B\ \ 30), B \leftarrow A, A \leftarrow t$ |
| Update chaining variables | $H_1 \leftarrow H_1 + A, H_2 \leftarrow H_2 + B,$<br>$H_3 \leftarrow H_3 + C, H_4 \leftarrow H_4 + D,$<br>$H_5 \leftarrow H_5 + E$ |

The incrementing of $N_1$, $N_2$, and $N_3$ during Rounds 0 and 1A is optional. A software implementation would not increment them, since it takes time, and at the end of the 16 times through the loop, all 4 counters will be their original values. Designers of hardware may wish to increment all 4 counters together to save on control logic. Round 0 can be completely omitted if the caller loads the 512 bits of $X_{0-15}$.

HMAC-SHA1

In the Authentication Chip implementation, the HMAC-SHA1 unit only ever performs hashing on two types of inputs: on R using $K_1$ and on R | M using $K_2$. Since the inputs are two constant lengths, rather than have HMAC and SHA-1 as separate entities on chip, they can be combined and the hardware optimized. The padding of messages in SHA-1 Step 1 (a 1 bit, a string of 0 bits, and the length of the message) is necessary to ensure that different messages will not look the same after padding. Since we only deal with 2 types of messages, our padding can be constant 0s. In addition, the optimized version of the SHA-1 algorithm is used, where only 16 32-bit words are used for temporary storage. These 16 registers are loaded directly by the optimized HMAC-SHA1 hardware. The Nine 32-bit constants $h_{1-5}$ and $y_{1-4}$ are still required, although the fact that they are constants is an advantage for hardware implementation. Hardware optimized HMAC-SHA-1 requires a total of 1024 bits of data storage:

Five 32-bit chaining variables are defined: $H_1$, $H_2$, $H_3$, $H_4$ and $H_5$.

Five 32-bit working variables are defined: A, B, C, D, and E.

Five 32-bit variables for temporary storage and final result: $Buff160_{1-5}$

One 32 bit temporary variable is defined: t.

Sixteen 32-bit temporary registers are defined: $X_{0-15}$.

The following two sections describe the steps for the two types of calls to HMAC-SHA1.

H[R, $K_1$]

In the case of producing the keyed hash of R using $K_1$, the original input message R is a constant length of 160 bits. We can therefore take advantage of this fact during processing. Rather than load $X_{0-15}$ during the first part of the SHA-1 algorithm, we load $X_{0-15}$ directly, and thereby omit Round 0 of the optimized Process Block (Step 2) of SHA-1. The pseudocode takes on the following steps:

| Step | Description | Action |
|---|---|---|
| 1 | Process K ⊕ ipad | $X_{0-4} \leftarrow K_1 \oplus 0x363636\ldots$ |
| 2 | | $X_{5-15} \leftarrow 0x363636\ldots$ |
| 3 | | $H_{1-5} \leftarrow h_{1-5}$ |
| 4 | | Process Block |
| | | |
| 5 | Process R | $X_{0-4} \leftarrow R$ |
| 6 | | $X_{5-15} \leftarrow 0$ |
| 7 | | Process Block |
| 8 | | $Buff160_{1-5} \leftarrow H_{1-5}$ |
| | | |
| 9 | Process K ⊕ opad | $X_{0-4} \leftarrow K_1 \oplus 0x5C5C5C\ldots$ |
| 10 | | $X_{5-15} \leftarrow 0x5C5C5C\ldots$ |
| 11 | | $H_{1-5} \leftarrow h_{1-5}$ |
| 12 | | Process Block |
| | | |
| 13 | Process previous H[x] | $X_{0-4} \leftarrow Result$ |
| 14 | | $X_{5-15} \leftarrow 0$ |
| 15 | | Process Block |
| | | |
| 16 | Get results | $Buff160_{1-5} \leftarrow H_{1-5}$ |

H[R | M, $K_2$]

In the case of producing the keyed hash of R | M using $K_2$, the original input message is a constant length of 416 (256+160) bits. We can therefore take advantage of this fact during processing. Rather than load $X_{0-15}$ during the first part of the SHA-1 algorithm, we load $X_{0-15}$ directly, and thereby omit Round 0 of the optimized Process Block (Step 2) of SHA-1. The pseudocode takes on the following steps:

| Step | Description | Action |
|---|---|---|
| 1 | Process K ⊕ ipad | $X_{0\text{-}4} \leftarrow K_2 \oplus$ 0x363636… |
| 2 | | $X_{5\text{-}15} \leftarrow$ 0x363636… |
| 3 | | $H_{1\text{-}5} \leftarrow h_{1\text{-}5}$ |
| 4 | | Process Block |
| | | |
| 5 | Process R \| M | $X_{0\text{-}4} \leftarrow R$ |
| 6 | | $X_{5\text{-}12} \leftarrow M$ |
| 7 | | $X_{13\text{-}15} \leftarrow 0$ |
| 8 | | Process Block |
| 9 | | Temp $\leftarrow H_{1\text{-}5}$ |
| | | |
| 10 | Process K ⊕ opad | $X_{0\text{-}4} \leftarrow K_2 \oplus$ 0x5C5C5C… |
| 11 | | $X_{5\text{-}15} \leftarrow$ 0x5C5C5C… |
| 12 | | $H_{1\text{-}5} \leftarrow h_{1\text{-}5}$ |
| 13 | | Process Block |
| | | |
| 14 | Process previous H[x] | $X_{0\text{-}4} \leftarrow$ Temp |
| 15 | | $X_{5\text{-}15} \leftarrow 0$ |
| 16 | | Process Block |
| | | |
| 17 | Get results | Result $\leftarrow H_{1\text{-}5}$ |

Data Storage Integrity

Each Authentication Chip contains some non-volatile memory in order to hold the variables required by Authentication Protocol 3. The following non-volatile variables are defined:

| Variable Name | Size (in bits) | Description |
|---|---|---|
| M[0..15] | 256 | 16 words (each 16 bits) containing state data such as serial numbers, media remaining etc. |
| $K_1$ | 160 | Key used to transform R during authentication. |
| $K_2$ | 160 | Key used to transform M during authentication. |
| R | 160 | Current random number |
| AccessMode[0..15] | 32 | The 16 sets of 2-bit AccessMode values for M[n]. |
| MinTicks | 32 | The minimum number of clock ticks between calls to key-based functions |
| SIWritten | 1 | If set, the secret key information ($K_1$, $K_2$, and R) has been written to the chip. If clear, the secret information has not been written yet. |
| IsTrusted | 1 | If set, the RND and TST functions can be called, but RD and WR functions cannot be called.<br>If clear, the RND and TST functions cannot be called, but RD and WR functions can be called. |
| **Total bits** | **802** | |

Note that if these variables are in Flash memory, it is not a simple matter to write a new value to replace the old. The memory must be erased first, and then the appropriate bits set. This has an effect on the algorithms used to change Flash memory based variables. For example, Flash memory cannot easily be used as shift registers. To update a Flash memory variable by a general operation, it is necessary to follow these steps:

Read the entire N bit value into a general purpose register;

Perform the operation on the general purpose register;

Erase the Flash memory corresponding to the variable; and

Set the bits of the Flash memory location based on the bits set in the general-purpose register.

A RESET of the Authentication Chip has no effect on these non-volatile variables.

M and AccessMode

Variables M[0] through M[15] are used to hold consumable state data, such as serial numbers, batch numbers, and amount of consumable remaining. Each M[n] register is 16 bits, making the entire M vector 256 bits (32 bytes). Clients cannot read from or written to individual M[n] variables. Instead, the entire vector, referred to as M, is read or written in a single logical access. M can be read using the RD (read) command, and written to via the WR (write) command. The commands only succeed if $K_1$ and $K_2$ are both defined (SIWritten = 1) and the Authentication Chip is a consumable non-trusted chip (IsTrusted = 0). Although M may contain a number of different data types, they differ only in their write permissions. Each data type can always be read. Once in client memory, the 256 bits can be

interpreted in any way chosen by the client. The entire 256 bits of M are read at one time instead of in smaller amounts for reasons of security, as described in the chapter entitled **Authentication**. The different write permissions are outlined in the following table:

| Data Type | Access Note |
|---|---|
| Read Only | Can never be written to |
| ReadWrite | Can always be written to |
| Decrement Only | Can only be written to if the new value is less than the old value. Decrement Only values are typically 16-bit or 32-bit values, but can be any multiple of 16 bits. |

To accomplish the protection required for writing, a 2-bit access mode value is defined for each M[n]. The following table defines the interpretation of the 2-bit access mode bit-pattern:

| Bits | Op | Interpretation | Action taken during Write command |
|---|---|---|---|
| 00 | RW | ReadWrite | The new 16-bit value is always written to M[n]. |
| 01 | MSR | Decrement Only (Most Significant Region) | The new 16-bit value is only written to M[n] if it is less than the value currently in M[n]. This is used for access to the Most Significant 16 bits of a Decrement Only number. |
| 10 | NMSR | Decrement Only (Not the Most Significant Region) | The new 16-bit value is only written to M[n] if M[n+1] can also be written. The NMSR access mode allows multiple precision values of 32 bits and more (multiples of 16 bits) to decrement. |
| 11 | RO | Read Only | The new 16-bit value is ignored. M[n] is left unchanged. |

The 16 sets of access mode bits for the 16 M[n] registers are gathered together in a single 32-bit AccessMode register. The 32 bits of the AccessMode register correspond to M[n] with n as follows:

MSB LSB

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Each 2-bit value is stored in hi/lo format. Consequently, if M[0-5] were access mode MSR, with M[6-15] access mode RO, the 32-bit AccessMode register would be:

11-11-11-11-11-11-11-11-11-11-01-01-01-01-01-01

During execution of a WR (write) command, AccessMode[n] is examined for each M[n], and a decision made as to whether the new M[n] value will replace the old. The AccessMode register is set using the Authentication Chip's SAM (Set Access Mode) command. Note that the Decrement Only comparison is unsigned, so any Decrement Only values that require negative ranges must be shifted into a positive range. For example, a consumable with a Decrement Only data item range of –50 to 50 must have the range shifted to be 0 to 100. The System must then interpret the range 0 to 100 as being –50 to 50. Note that most instances of Decrement Only ranges are N to 0, so there is no range shift required. For Decrement Only data items, arrange the data in order from most significant to least significant 16-bit quantities from M[n] onward. The access mode for the most significant 16 bits (stored in M[n]) should be set to MSR. The remaining registers (M[n+1], M[n+2] etc) should have their access modes set to NMSR. If erroneously set to NMSR, with no associated MSR region, each NMSR region will be considered independently instead of being a multi-precision comparison.

<u>$K_1$</u>

$K_1$ is the 160-bit secret key used to transform R during the authentication protocol. $K_1$ is programmed along with $K_2$ and R with the SSI (Set Secret Information) command. Since $K_1$ must be kept secret, clients cannot directly read $K_1$. The commands that make use of $K_1$ are RND and RD. RND returns a pair R, $F_{K1}[R]$ where R is a random number, while RD requires an X, $F_{K1}[X]$ pair as input. $K_1$ is used in the keyed one-way hash function HMAC-SHA1. As such it should be programmed with a physically generated random number, gathered from a physically random phenomenon. **$K_1$ must NOT be generated with a computer-run random number generator**. The security of the Authentication chips depends on $K_1$, $K_2$ and R being generated in a way that is not deterministic. For example, to set $K_1$, a person can toss a fair coin 160 times, recording heads as 1, and tails as 0. $K_1$ is automatically cleared to 0 upon execution of a CLR command. It can only be programmed to a non-zero value by the SSI command.

<u>$K_2$</u>

$K_2$ is the 160-bit secret key used to transform M | R during the authentication protocol. $K_2$ is programmed along with $K_1$ and R with the SSI (Set Secret Information) command. Since $K_2$ must be kept secret, clients cannot directly read $K_2$. The commands that make use of $K_2$ are RD and TST. RD returns a pair M, $F_{K2}[M | X]$ where X was passed in as one of the parameters to the RD function. TST requires an M, $F_{K2}[M | R]$ pair as input, where R was obtained from the Authentication Chip's RND function. $K_2$ is used in the keyed one-way hash function HMAC-SHA1. As such it should be programmed with a physically generated random number, gathered from a physically random phenomenon. **$K_2$ must NOT be generated with a computer-run random number generator**. The security of the Authentication chips depends on $K_1$, $K_2$ and R being generated in a way that is not deterministic. For example, to set $K_2$, a person can toss a fair coin 160 times, recording heads as 1, and tails as 0. $K_2$ is automatically cleared to 0 upon execution of a CLR command. It can only be programmed to a non-zero value by the SSI command.

<u>R and IsTrusted</u>

R is a 160-bit random number seed that is programmed along with $K_1$ and $K_2$ with the SSI (Set Secret Information) command. R does not have to be kept secret, since it is given freely to callers via the RND command. However R must be changed only by the Authentication Chip, and not set to any chosen value by a caller. R is used during the TST command to ensure that the R from the previous call to RND was used to generate the $F_{K2}[M | R]$ value in the non-trusted Authentication Chip (ChipA). Both RND and TST are only used in trusted Authentication Chips (ChipT).

IsTrusted is a 1-bit flag register that determines whether or not the Authentication Chip is a trusted chip (ChipT):

If the IsTrusted bit is set, the chip is considered to be a trusted chip, and hence clients can call RND and TST functions (but not RD or WR).

If the IsTrusted bit is clear, the chip is not considered to be trusted. Therefore RND and TST functions cannot be called (but RD and WR functions can be called instead). System never needs to call RND or TST on the consumable (since a clone chip would simply return 1 to a function such as TST, and a constant value for RND).

The IsTrusted bit has the added advantage of reducing the number of available R, $F_{K1}[R]$ pairs obtainable by an attacker, yet still maintain the integrity of the Authentication protocol. To obtain valid R, $F_{K1}[R]$ pairs, an attacker requires a System Authentication Chip, which is more expensive and less readily available than the consumables. Both R and the IsTrusted bit are cleared to 0 by the CLR command. They are both written to by the issuing of the SSI command. The IsTrusted bit can only set by storing a non-zero seed value in R via the SSI command (R must be non-zero to be a valid LFSR state, so this is quite reasonable). R is changed via a 160-bit maximal period LFSR with taps on bits 1, 2, 4, and 159, and is changed only by a successful call to TST (where 1 is returned).

Authentication Chips destined to be trusted Chips used in Systems (ChipT) should have their IsTrusted bit set during programming, and Authentication Chips used in Consumables (ChipA) should have their IsTrusted bit kept clear (by storing 0 in R via the SSI command during programming). There is no command to read or write the IsTrusted bit directly. The security of the Authentication Chip does not only rely upon the randomness of $K_1$ and $K_2$ and the strength of the HMAC-SHA1 algorithm. To prevent an attacker from building a sparse lookup table, the security of the Authentication Chip also depends on the range of R over the lifetime of all Systems. What this means is that an attacker must not be able to deduce what values of R there are in produced and future Systems. As such R should be programmed with a physically generated random number, gathered from a physically random phenomenon. **R must NOT be generated with a computer-run random number generator**. The generation of R must not be deterministic. For example, to generate an R for use in a trusted System chip, a person can toss a fair coin 160 times, recording heads as 1, and tails as 0. 0 is the only non-valid initial value for a trusted R is 0 (or the IsTrusted bit will not be set).

SIWritten

The SIWritten (Secret Information **Written**) 1-bit register holds the status of the secret information stored within the Authentication Chip. The secret information is $K_1$, $K_2$ and R. A client cannot directly access the SIWritten bit. Instead, it is cleared via the CLR command (which also clears $K_1$, $K_2$ and R). When the Authentication Chip is programmed with secret keys and random number seed using the SSI command (regardless of the value written), the SIWritten bit is set automatically. Although R is strictly not secret, it must be written together with $K_1$ and $K_2$ to ensure that an attacker cannot generate their own random number seed in order to obtain chosen R, $F_{K1}[R]$ pairs. The SIWritten status bit is used by all functions that access $K_1$, $K_2$, or R. If the SIWritten bit is clear, then calls to RD, WR, RND, and TST are interpreted as calls to CLR.

MinTicks

There are two mechanisms for preventing an attacker from generating multiple calls to TST and RD functions in a short period of time. The first is a clock limiting hardware component that prevents the internal clock from operating at

a speed more than a particular maximum (e.g. 10 MHz). The second mechanism is the 32-bit MinTicks register, which is used to specify the minimum number of clock ticks that must elapse between calls to key-based functions. The MinTicks variable is cleared to 0 via the CLR command. Bits can then be set via the SMT (Set MinTicks) command. The input parameter to SMT contains the bit pattern that represents which bits of MinTicks are to be set. The practical effect is that an attacker can only increase the value in MinTicks (since the SMT function only sets bits). In addition, there is no function provided to allow a caller to read the current value of this register. The value of MinTicks depends on the operating clock speed and the notion of what constitutes a reasonable time between key-based function calls (application specific). The duration of a single tick depends on the operating clock speed. This is the maximum of the input clock speed and the Authentication Chip's clock-limiting hardware. For example, the Authentication Chip's clock-limiting hardware may be set at 10 MHz (it is not changeable), but the input clock is 1 MHz. In this case, the value of 1 tick is based on 1 MHz, not 10 MHz. If the input clock was 20 MHz instead of 1 MHz, the value of 1 tick is based on 10 MHz (since the clock speed is limited to 10 MHz).

Once the duration of a tick is known, the MinTicks value can to be set. The value for MinTicks is the minimum number of ticks required to pass between calls to the key-based RD and TST functions. The value is a real-time number, and divided by the length of an operating tick. Suppose the input clock speed matches the maximum clock speed of 10 MHz. If we want a minimum of 1 second between calls to key based functions, the value for MinTicks is set to 10,000,000. Consider an attacker attempting to collect X, $F_{K1}[X]$ pairs by calling RND, RD and TST multiple times. If the MinTicks value is set such that the amount of time between calls to TST is 1 second, then each pair requires 1 second to generate. To generate $2^{25}$ pairs (only requiring 1.25 GB of storage), an attacker requires more than 1 year. An attack requiring $2^{64}$ pairs would require $5.84 \times 10^{11}$ years using a single chip, or 584 years if 1 billion chips were used, making such an attack completely impractical in terms of time (not to mention the storage requirements!).

With regards to $K_1$, it should be noted that the MinTicks variable only slows down an attacker and causes the attack to cost more since it does not stop an attacker using multiple System chips in parallel. However MinTicks does make an attack on $K_2$ more difficult, since each consumable has a different M (part of M is random read-only data). In order to launch a differential attack, minimally different inputs are required, and this can only be achieved with a single consumable (containing an effectively constant part of M). Minimally different inputs require the attacker to use a single chip, and MinTicks causes the use of a single chip to be slowed down. If it takes a year just to get the data to start searching for values to begin a differential attack this increases the cost of attack and reduces the effective market time of a clone consumable.

Authentication Chip Commands

The System communicates with the Authentication Chips via a simple operation command set. This section details the actual commands and parameters necessary for implementation of Protocol 3. The Authentication Chip is defined here as communicating to System via a serial interface as a minimum implementation. It is a trivial matter to define an equivalent chip that operates over a wider interface (such as 8, 16 or 32 bits). Each command is defined by 3-bit opcode. The interpretation of the opcode can depend on the current value of the IsTrusted bit and the current value of the IsWritten bit. The following operations are defined:

| Op | T | W | Mn | Input | Output | Description |
|---|---|---|---|---|---|---|
| 000 | - | - | CLR | - | - | Clear |
| 001 | 0 | 0 | SSI | [160, 160, 160] | - | Set Secret Information |
| 010 | 0 | 1 | RD | [160, 160] | [256, 160] | Read M securely |
| 010 | 1 | 1 | RND | - | [160, 160] | Random |
| 011 | 0 | 1 | WR | [256] | - | Write M |
| 011 | 1 | 1 | TST | [256, 160] | [1] | Test |
| 100 | 0 | 1 | SAM | [32] | [32] | Set Access Mode |
| 101 | - | 1 | GIT | - | [1] | Get Is Trusted |
| 110 | - | 1 | SMT | [32] | - | Set MinTicks |

**Op** = Opcode, **T** = IsTrusted value, **W** = IsWritten value,
**Mn** = Mnemonic, [n] = number of bits required for parameter

Any command not defined in this table is interpreted as NOP (**No Operation**). Examples include opcodes 110 and 111 (regardless of IsTrusted or IsWritten values), and any opcode other than SSI when IsWritten = 0. Note that the opcodes for RD and RND are the same, as are the opcodes for WR and TST. The actual command run upon receipt of the opcode will depend on the current value of the IsTrusted bit (as long as IsWritten is 1). Where the IsTrusted bit is clear, RD and WR functions will be called. Where the IsTrusted bit is set, RND and TST functions will be called. The two sets of commands are mutually exclusive between trusted and non-trusted Authentication Chips, and the same opcodes enforces this relationship. Each of the commands is examined in detail in the subsequent sections. Note that some algorithms are specifically designed because Flash memory is assumed for the implementation of non-volatile variables.

| **CLR** | **Clear** |
|---|---|
| **Input** | None |
| **Output** | None |
| **Changes** | All |

The CLR (**Clear**) Command is designed to completely erase the contents of all Authentication Chip memory. This includes all keys and secret information, access mode bits, and state data. After the execution of the CLR command, an Authentication Chip will be in a programmable state, just as if it had been freshly manufactured. It can be reprogrammed with a new key and reused. A CLR command consists of simply the CLR command opcode. Since the Authentication Chip is serial, this must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A CLR command is therefore sent as bits 0-2 of the CLR opcode. A total of 3 bits are transferred. The CLR command can be called directly at any time. The order of erasure is important. SIWritten must

be cleared first, to disable further calls to key access functions (such as RND, TST, RD and WR). If the AccessMode bits are cleared before SIWritten, an attacker could remove power at some point after they have been cleared, and manipulate M, thereby have a better chance of retrieving the secret information with a partial chosen text attack. The CLR command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | Erase SIWritten<br>Erase IsTrusted<br>Erase $K_1$<br>Erase $K_2$<br>Erase R<br>Erase M |
| 2 | Erase AccessMode<br>Erase MinTicks |

Once the chip has been cleared it is ready for reprogramming and reuse. A blank chip is of no use to an attacker, since although they can create any value for M (M can be read from and written to), key-based functions will not provide any information as $K_1$ and $K_2$ will be incorrect. It is not necessary to consume any input parameter bits if CLR is called for any opcode other than CLR. An attacker will simply have to RESET the chip. The reason for calling CLR is to ensure that all secret information has been destroyed, making the chip useless to an attacker.

SSI – Set Secret Information

**Input:** $K_1$, $K_2$, R = [160 bits, 160 bits, 160 bits]

**Output:** None

**Changes:** $K_1$, $K_2$, R, SIWritten, IsTrusted

The SSI (Set Secret Information) command is used to load the $K_1$, $K_2$ and R variables, and to set SIWritten and IsTrusted flags for later calls to RND, TST, RD and WR commands. An SSI command consists of the SSI command opcode followed by the secret information to be stored in the $K_1$, $K_2$ and R registers. Since the Authentication Chip is serial, this must be transferred one bit at a time. The bit order is LSB to MSB for each command component. An SSI command is therefore sent as: bits 0-2 of the SSI opcode, followed by bits 0-159 of the new value for $K_1$, bits 0-159 of the new value for $K_2$, and finally bits 0-159 of the seed value for R. A total of 483 bits are transferred. The $K_1$, $K_2$, R, SIWritten, and IsTrusted registers are all cleared to 0 with a CLR command. They can only be set using the SSI command.

The SSI command uses the flag SIWritten to store the fact that data has been loaded into $K_1$, $K_2$, and R. If the SIWritten and IsTrusted flags are clear (this is the case after a CLR instruction), then $K_1$, $K_2$ and R are loaded with the new values. If either flag is set, an attempted call to SSI results in a CLR command being executed, since only an

attacker or an erroneous client would attempt to change keys or the random seed without calling CLR first. The SSI command also sets the IsTrusted flag depending on the value for R. If R = 0, then the chip is considered untrustworthy, and therefore IsTrusted remains at 0. If R ≠ 0, then the chip is considered trustworthy, and therefore IsTrusted is set to 1. Note that the setting of the IsTrusted bit only occurs during the SSI command. If an Authentication Chip is to be reused, the CLR command must be called first. The keys can then be safely reprogrammed with an SSI command, and fresh state information loaded into M using the SAM and WR commands. The SSI command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | CLR |
| 2 | $K_1 \leftarrow$ Read 160 bits from client |
| 3 | $K_2 \leftarrow$ Read 160 bits from client |
| 4 | $R \leftarrow$ Read 160 bits from client |
| 5 | IF ($R \neq 0$)<br>IsTrusted $\leftarrow$ 1 |
| 6 | SIWritten $\leftarrow$ 1 |

RD – Read

**Input:** X, $F_{K1}[X]$ = [160 bits, 160 bits]

**Output:** M, $F_{K2}[X \mid M]$ = [256 bits, 160 bits]

**Changes:** R

The RD (**Read**) command is used to securely read the entire 256 bits of state data (M) from a non-trusted Authentication Chip. Only a valid Authentication Chip will respond correctly to the RD request. The output bits from the RD command can be fed as the input bits to the TST command on a trusted Authentication Chip for verification, with the first 256 bits (M) stored for later use if (as we hope) TST returns 1. Since the Authentication Chip is serial, the command and input parameters must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A RD command is therefore: bits 0-2 of the RD opcode, followed by bits 0-159 of X, and bits 0-159 of $F_{K1}[X]$. 323 bits are transferred in total. X and $F_{K1}[X]$ are obtained by calling the trusted Authentication Chip's RND command. The 320 bits output by the trusted chip's RND command can therefore be fed directly into the non-trusted chip's RD command, with no need for these bits to be stored by System. The RD command can only be used when the following conditions have been met:

| | |
|---|---|
| SIWritten = 1 | indicating that $K_1$, $K_2$ and R have been set up via the SSI command; and |
| IsTrusted = 0 | indicating the chip is not trusted since it is not permitted to generate random number sequences; |

In addition, calls to RD must wait for the MinTicksRemaining register to reach 0. Once it has done so, the register is reloaded with MinTicks to ensure that a minimum time will elapse between calls to RD. Once MinTicksRemaining has been reloaded with MinTicks, the RD command verifies that the input parameters are valid. This is accomplished by

internally generating $F_{K1}[X]$ for the input X, and then comparing the result against the input $F_{K1}[X]$. This generation and comparison must take the same amount of time regardless of whether the input parameters are correct or not. If the times are not the same, an attacker can gain information about which bits of $F_{K1}[X]$ are incorrect. The only way for the input parameters to be invalid is an erroneous System (passing the wrong bits), a case of the wrong consumable in the wrong System, a bad trusted chip (generating bad pairs), or an attack on the Authentication Chip. A constant value of 0 is returned when the input parameters are wrong. The time taken for 0 to be returned must be the same for all bad inputs so that attackers can learn nothing about what was invalid. Once the input parameters have been verified the output values are calculated. The 256 bit content of M are transferred in the following order: bits 0-15 of M[0], bits 0-15 of M[1], through to bits 0-15 of M[15]. $F_{K2}[X \mid M]$ is calculated and output as bits 0-159. The R register is used to store the X value during the validation of the X, $F_{K1}[X]$ pair. This is because RND and RD are mutually exclusive. The RD command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | IF (MinTicksRemaining ≠ 0<br>GOTO 1 |
| 2 | MinTicksRemaining ← MinTicks |
| 3 | R ← Read 160 bits from client |
| 4 | Hash ← Calculate $F_{K1}[R]$ |
| 5 | OK ← (Hash = next 160 bits from client)<br>Note that this operation must take constant time so an attacker cannot determine how much of their guess is correct. |
| 6 | IF (OK)<br>Output 256 bits of M to client<br>ELSE<br>Output 256 bits of 0 to client |
| 7 | Hash ← Calculate $F_{K2}[R \mid M]$ |
| 8 | IF (OK)<br>Output 160 bits of Hash to client<br>ELSE<br>Output 160 bits of 0 to client |

<u>RND – Random</u>

**Input:** None

**Output:** R, $F_{K1}[R]$ = [160 bits, 160 bits]

**Changes:** None

The RND (**Random**) command is used by a client to obtain a valid R, $F_{K1}[R]$ pair for use in a subsequent authentication via the RD and TST commands. Since there are no input parameters, an RND command is therefore simply bits 0-2 of the RND opcode. The RND command can only be used when the following conditions have been met:

| | |
|---|---|
| SIWritten = 1 | indicating $K_1$ and R have been set up via the SSI command; |
| IsTrusted = 1 | indicating the chip is permitted to generate random number sequences; |

RND returns both R and $F_{K1}[R]$ to the caller. The 288-bit output of the RND command can be fed straight into the non-trusted chip's RD command as the input parameters. There is no need for the client to store them at all, since they are not required again. However the TST command will only succeed if the random number passed into the RD command was obtained first from the RND command. If a caller only calls RND multiple times, the same R, $F_{K1}[R]$ pair will be returned each time. R will only advance to the next random number in the sequence after a successful call to TST. See TST for more information. The RND command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | Output 160 bits of R to client |
| 2 | Hash ← Calculate $F_{K1}[R]$ |
| 3 | Output 160 bits of Hash to client |

TST – Test

**Input:** X, $F_{K2}[R \mid X]$ = [256 bits, 160 bits]

**Output:** 1 or 0 = [1 bit]

**Changes:** M, R and MinTicksRemaining (or all registers if attack detected)

The TST (**Test**) command is used to authenticate a read of M from a non-trusted Authentication Chip. The TST (**Test**) command consists of the TST command opcode followed by input parameters: X and $F_{K2}[R \mid X]$. Since the Authentication Chip is serial, this must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A TST command is therefore: bits 0-2 of the TST opcode, followed by bits 0-255 of M, bits 0-159 of $F_{K2}[R \mid M]$. 419 bits are transferred in total. Since the last 416 input bits are obtained as the output bits from a RD command to a non-trusted Authentication Chip, the entire data does not even have to be stored by the client. Instead, the bits can be passed directly to the trusted Authentication Chip's TST command. Only the 256 bits of M should be kept from a RD command. The TST command can only be used when the following conditions have been met:

| | |
|---|---|
| SIWritten = 1 | indicating $K_2$ and R have been set up via the SSI command; |
| IsTrusted = 1 | indicating the chip is permitted to generate random number sequences; |

In addition, calls to TST must wait for the MinTicksRemaining register to reach 0. Once it has done so, the register is reloaded with MinTicks to ensure that a minimum time will elapse between calls to TST. TST causes the internal M value to be replaced by the input M value. $F_{K2}[M \mid R]$ is then calculated, and compared against the 160 bit input hash value. A single output bit is produced: 1 if they are the same, and 0 if they are different. The use of the internal M value is to save space on chip, and is the reason why RD and TST are mutually exclusive commands. If the output bit is 1, R

is updated to be the next random number in the sequence. This forces the caller to use a new random number each time RD and TST are called. The resultant output bit is not output until the entire input string has been compared, so that the time to evaluate the comparison in the TST function is **always** the same. Thus no attacker can compare execution times or number of bits processed before an output is given.

The next random number is generated from R using a 160-bit maximal period LFSR (tap selections on bits 159, 4, 2, and 1). The initial 160-bit value for R is set up via the SSI command, and can be any random number except 0 (an LFSR filled with 0s will produce a never-ending stream of 0s). R is transformed by XORing bits 1, 2, 4, and 159 together, and shifting all 160 bits right 1 bit using the XOR result as the input bit to $b_{159}$. The new R will be returned on the next call to RND. Note that the time taken for 0 to be returned from TST must be the same for all bad inputs so that attackers can learn nothing about what was invalid about the input.

The TST command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | IF (MinTicksRemaining ≠ 0<br>GOTO 1 |
| 2 | MinTicksRemaining ← MinTicks |
| 3 | M ← Read 256 bits from client |
| 4 | IF (R = 0)<br>GOTO CLR |
| 5 | Hash ← Calculate $F_{K2}[R \mid M]$ |
| 6 | OK ← (Hash = next 160 bits from client)<br>Note that this operation must take constant time so an attacker cannot determine how much of their guess is correct. |
| 7 | IF (OK)<br>Temp ← R<br>Erase R<br>Advance TEMP via LFSR<br>R ← TEMP |
| 8 | Output 1 bit of OK to client |

Note that we can't simply advance R directly in Step 7 since R is Flash memory, and must be erased in order for any set bit to become 0. If power is removed from the Authentication Chip during Step 7 after erasing the old value of R, but before the new value for R has been written, then R will be erased but not reprogrammed. We therefore have the situation of IsTrusted=1, yet R=0, a situation only possible due to an attacker. Step 4 detects this event, and takes action if the attack is detected. This problem can be avoided by having a second 160-bit Flash register for R and a

Validity Bit, toggled after the new value has been loaded. It has not been included in this implementation for reasons of space, but if chip space allows it, an extra 160-bit Flash register would be useful for this purpose.

WR – Write

**Input:** $M_{new}$ = [256 bits]

**Output:** None

**Changes:** M

A WR (Write) command is used to update the writeable parts of M containing Authentication Chip state data. The WR command by itself is not secure. It must be followed by an authenticated read of M (via a RD command) to ensure that the change was made as specified. The WR command is called by passing the WR command opcode followed by the new 256 bits of data to be written to M. Since the Authentication Chip is serial, the new value for M must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A WR command is therefore: bits 0-2 of the WR opcode, followed by bits 0-15 of M[0], bits 0-15 of M[1], through to bits 0-15 of M[15]. 259 bits are transferred in total. The WR command can only be used when SIWritten = 1, indicating that $K_1$, $K_2$ and R have been set up via the SSI command (if SIWritten is 0, then $K_1$, $K_2$ and R have not been setup yet, and the CLR command is called instead). The ability to write to a specific M[n] is governed by the corresponding Access Mode bits as stored in the AccessMode register. The AccessMode bits can be set using the SAM command. When writing the new value to M[n] the fact that M[n] is Flash memory must be taken into account. All the bits of M[n] must be erased, and then the appropriate bits set. Since these two steps occur on different cycles, it leaves the possibility of attack open. An attacker can remove power after erasure, but before programming with the new value. However, there is no advantage to an attacker in doing this:

A Read/Write M[n] changed to 0 by this means is of no advantage since the attacker could have written any value using the WR command anyway.

A Read Only M[n] changed to 0 by this means allows an additional known text pair (where the M[n] is 0 instead of the original value). For future use M[n] values, they are already 0, so no information is given.

A Decrement Only M[n] changed to 0 simply speeds up the time in which the consumable is used up. It does not give any new information to an attacker that using the consumable would give.

The WR command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | DecEncountered ← 0<br>EqEncountered ← 0<br>n ← 15 |
| 2 | Temp ← Read 16 bits from client |
| 3 | AM = AccessMode[~n] |
| Compare to the previous value | |
| 5 | LT ← (Temp < M[~n]) [comparison is unsigned]<br>EQ ← (Temp = M[~n]) |
| 6 | WE ← (AM = RW) ∨<br>((AM = MSR) ∧ LT) ∨<br>((AM = NMSR) ∧ (DecEncountered ∨ LT)) |
| 7 | DecEncountered ← ((AM = MSR) ∧ LT) ∨<br>((AM = NMSR) ∧ DecEncountered) ∨<br>((AM = NMSR) ∧ EqEncountered ∧ LT)<br>EqEncountered ← ((AM = MSR) ∧ EQ) ∨<br>((AM = NMSR) ∧ EqEncountered ∧ EQ) |
| Advance to the next Access Mode set and write the new M[~n] if applicable | |
| 8 | IF (WE)<br>Erase M[~n]<br>M[~n] ← Temp |
| 10 | ⇓n |
| 11 | IF (n ≠ 0)<br>GOTO 2 |

SAM – Set AccessMode

**Input:** $AccessMode_{new}$ = [32 bits]

**Output:** AccessMode = [32 bits]

**Changes:** AccessMode

The SAM (Set Access Mode) command is used to set the 32 bits of the AccessMode register, and is only available for

use in consumable Authentication Chips (where the IsTrusted flag = 0). The SAM command is called by passing the SAM command opcode followed by a 32-bit value that is used to set bits in the AccessMode register. Since the Authentication Chip is serial, the data must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A SAM command is therefore: bits 0-2 of the SAM opcode, followed by bits 0-31 of bits to be set in AccessMode. 35 bits are transferred in total. The AccessMode register is only cleared to 0 upon execution of a CLR command. Since an access mode of 00 indicates an access mode of RW (read/write), not setting any AccessMode bits after a CLR means that all of M can be read from and written to. The SAM command only sets bits in the AccessMode register. Consequently a client can change the access mode bits for M[n] from RW to RO (read only) by setting the appropriate bits in a 32-bit word, and calling SAM with that 32-bit value as the input parameter. This allows the programming of the access mode bits at different times, perhaps at different stages of the manufacturing process. For example, the read only random data can be written to during the initial key programming stage, while allowing a second programming stage for items such as consumable serial numbers.

Since the SAM command only sets bits, the effect is to allow the access mode bits corresponding to M[n] to progress from RW to either MSR, NMSR, or RO. It should be noted that an access mode of MSR can be changed to RO, but this would not help an attacker, since the authentication of M after a write to a doctored Authentication Chip would detect that the write was not successful and hence abort the operation. The setting of bits corresponds to the way that Flash memory works best. The only way to clear bits in the AccessMode register, for example to change a Decrement Only M[n] to be Read/Write, is to use the CLR command. The CLR command not only erases (clears) the AccessMode register, but also clears the keys and all of M. Thus the AccessMode[n] bits corresponding to M[n] can only usefully be changed once between CLR commands. The SAM command returns the new value of the AccessMode register (after the appropriate bits have been set due to the input parameter). By calling SAM with an input parameter of 0, AccessMode will not be changed, and therefore the current value of AccessMode will be returned to the caller.

The SAM command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | Temp ← Read 32 bits from client |
| 2 | SetBits(AccessMode, Temp) |
| 3 | Output 32 bits of AccessMode to client |

GIT – Get Is Trusted

**Input:** None

**Output:** IsTrusted = [1 bit]

**Changes:** None

The GIT (Get Is Trusted) command is used to read the current value of the IsTrusted bit on the Authentication Chip. If the bit returned is 1, the Authentication Chip is a trusted System Authentication Chip. If the bit returned is 0, the Authentication Chip is a consumable Authentication Chip. A GIT command consists of simply the GIT command

opcode. Since the Authentication Chip is serial, this must be transferred one bit at a time. The bit order is LSB to MSB for each command component. A GIT command is therefore sent as bits 0-2 of the GIT opcode. A total of 3 bits are transferred. The GIT command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | Output IsTrusted bit to client |

SMT – Set MinTicks

**Input:** $MinTicks_{new}$ = [32 bits]

**Output:** None

**Changes:** MinTicks

The SMT (Set MinTicks) command is used to set bits in the MinTicks register and hence define the minimum number of ticks that must pass in between calls to TST and RD. The SMT command is called by passing the SMT command opcode followed by a 32-bit value that is used to set bits in the MinTicks register. Since the Authentication Chip is serial, the data must be transferred one bit at a time. The bit order is LSB to MSB for each command component. An SMT command is therefore: bits 0-2 of the SMT opcode, followed by bits 0-31 of bits to be set in MinTicks. 35 bits are transferred in total. The MinTicks register is only cleared to 0 upon execution of a CLR command. A value of 0 indicates that no ticks need to pass between calls to key-based functions. The functions may therefore be called as frequently as the clock speed limiting hardware allows the chip to run.

Since the SMT command only sets bits, the effect is to allow a client to set a value, and only increase the time delay if further calls are made. Setting a bit that is already set has no effect, and setting a bit that is clear only serves to slow the chip down further. The setting of bits corresponds to the way that Flash memory works best. The only way to clear bits in the MinTicks register, for example to change a value of 10 ticks to a value of 4 ticks, is to use the CLR command. However the CLR command clears the MinTicks register to 0 as well as clearing all keys and M. It is therefore useless for an attacker. Thus the MinTicks register can only usefully be changed once between CLR commands.

The SMT command is implemented with the following steps:

| Step | Action |
|---|---|
| 1 | Temp ← Read 32 bits from client |
| 2 | SetBits(MinTicks, Temp) |

Programming Authentication Chips

Authentication Chips must be programmed with logically secure information in a physically secure environment. Consequently the programming procedures cover both logical and physical security. Logical security is the process of

ensuring that $K_1$, $K_2$, R, and the random M[n] values are generated by a physically random process, and not by a computer. It is also the process of ensuring that the order in which parts of the chip are programmed is the most logically secure. Physical security is the process of ensuring that the programming station is physically secure, so that $K_1$ and $K_2$ remain secret, both during the key generation stage and during the lifetime of the storage of the keys. In addition, the programming station must be resistant to physical attempts to obtain or destroy the keys. The Authentication Chip has its own security mechanisms for ensuring that $K_1$ and $K_2$ are kept secret, but the Programming Station must also keep $K_1$ and $K_2$ safe.

Overview

After manufacture, an Authentication Chip must be programmed before it can be used. In all chips values for $K_1$ and $K_2$ must be established. If the chip is destined to be a System Authentication Chip, the initial value for R must be determined. If the chip is destined to be a consumable Authentication Chip, R must be set to 0, and initial values for M and AccessMode must be set up. The following stages are therefore identified:

Determine Interaction between Systems and Consumables

Determine Keys for Systems and Consumables

Determine MinTicks for Systems and Consumables

Program Keys, Random Seed, MinTicks and Unused M

Program State Data and Access Modes

Once the consumable or system is no longer required, the attached Authentication Chip can be reused. This is easily accomplished by reprogrammed the chip starting at Stage 4 again. Each of the stages is examined in the subsequent sections.

Stage 0: Manufacture

The manufacture of Authentication Chips does not require any special security. There is no secret information programmed into the chips at manufacturing stage. The algorithms and chip process is not special. Standard Flash processes are used. A theft of Authentication Chips between the chip manufacturer and programming station would only provide the clone manufacturer with blank chips. This merely compromises the sale of Authentication chips, not anything authenticated by Authentication Chips. Since the programming station is the only mechanism with consumable and system product keys, a clone manufacturer would not be able to program the chips with the correct key. Clone manufacturers would be able to program the blank chips for their own systems and consumables, but it would be difficult to place these items on the market without detection. In addition, a single theft would be difficult to base a business around.

Stage 1: Determine Interaction between Systems and Consumables

The decision of what is a System and what is a Consumable needs to be determined before any Authentication Chips can be programmed. A decision needs to be made about which Consumables can be used in which Systems, since all connected Systems and Consumables must share the same key information. They also need to share state-data usage mechanisms even if some of the interpretations of that data have not yet been determined. A simple example is that of a car and car-keys. The car itself is the System, and the car-keys are the consumables. There are several car-keys for each car, each containing the same key information as the specific car. However each car (System) would contain a different key (shared by its car-keys), since we don't want car-keys from one car working in another. Another example

is that of a photocopier that requires a particular toner cartridge. In simple terms the photocopier is the System, and the toner cartridge is the consumable. However the decision must be made as to what compatibility there is to be between cartridges and photocopiers. The decision has historically been made in terms of the physical packaging of the toner cartridge: certain cartridges will or won't fit in a new model photocopier based on the design decisions for that copier. When Authentication Chips are used, the components that must work together must share the same key information.

In addition, each type of consumable requires a different way of dividing M (the state data). Although the way in which M is used will vary from application to application, the method of allocating M[n] and AccessMode[n] will be the same:

Define the consumable state data for specific use

Set some M[n] registers aside for future use (if required). Set these to be 0 and Read Only. The value can be tested for in Systems to maintain compatibility.

Set the remaining M[n] registers (at least one, but it does not have to be M[15]) to be Read Only, with the contents of each M[n] completely random. This is to make it more difficult for a clone manufacturer to attack the authentication keys.

The following examples show ways in which the state data may be organized.

Example 1

Suppose we have a car with associated car-keys. A 16-bit key number is more than enough to uniquely identify each car-key for a given car. The 256 bits of M could be divided up as follows:

| M[n] | Access | Description |
|---|---|---|
| 0 | RO | Key number (16 bits) |
| 1-4 | RO | Car engine number (64 bits) |
| 5-8 | RO | For future expansion = 0 (64 bits) |
| 8-15 | RO | Random bit data (128 bits) |

If the car manufacturer keeps all logical keys for all cars, it is a trivial matter to manufacture a new physical car-key for a given car should one be lost. The new car-key would contain a new Key Number in M[0], but have the same $K_1$ and $K_2$ as the car's Authentication Chip. Car Systems could allow specific key numbers to be invalidated (for example if a key is lost). Such a system might require Key 0 (the master key) to be inserted first, then all valid keys, then Key 0 again. Only those valid keys would now work with the car. In the worst case, for example if all car-keys are lost, then a new set of logical keys could be generated for the car and its associated physical car-keys if desired. The Car engine number would be used to tie the key to the particular car. Future use data may include such things as rental information, such as driver/renter details.

Example 2

Suppose we have a photocopier image unit which should be replaced every 100,000 copies. 32 bits are required to store the number of pages remaining. The 256 bits of M could be divided up as follows:

| M[n] | Access | Description |
|---|---|---|

| | | |
|---|---|---|
| 0 | RO | Serial number (16 bits) |
| 1 | RO | Batch number (16 bits) |
| 2 | MSR | Page Count Remaining (32 bits, hi/lo) |
| 3 | NMSR | |
| 4-7 | RO | For future expansion = 0 (64 bits) |
| 8-15 | RO | Random bit data (128 bits) |

If a lower quality image unit is made that must be replaced after only 10,000 copies, the 32-bit page count can still be used for compatibility with existing photocopiers. This allows several consumable types to be used with the same system.

Example 3

Consider a Polaroid camera consumable containing 25 photos. A 16-bit countdown is all that is required to store the number of photos remaining. The 256 bits of M could be divided up as follows:

| M[n] | Access | Description |
|---|---|---|
| 0 | RO | Serial number (16 bits) |
| 1 | RO | Batch number (16 bits) |
| 2 | MSR | Photos Remaining (16 bits) |
| 3-6 | RO | For future expansion = 0 (64 bits) |
| 7-15 | RO | Random bit data (144 bits) |

The Photos Remaining value at M[2] allows a number of consumable types to be built for use with the same camera System. For example, a new consumable with 36 photos is trivial to program. Suppose 2 years after the introduction of the camera, a new type of camera was introduced. It is able to use the old consumable, but also can process a new film type. M[3] can be used to define Film Type. Old film types would be 0, and the new film types would be some new value. New Systems can take advantage of this. Original systems would detect a non-zero value at M[3] and realize incompatibility with new film types. New Systems would understand the value of M[3] and so react appropriately. To maintain compatibility with the old consumable, the new consumable and System needs to have the same key information as the old one. To make a clean break with a new System and its own special consumables, a new key set would be required.

Example 4

Consider a printer consumable containing 3 inks: cyan, magenta, and yellow. Each ink amount can be decremented separately. The 256 bits of M could be divided up as follows:

| M[n] | Access | Description |
|---|---|---|
| 0 | RO | Serial number (16 bits) |
| 1 | RO | Batch number (16 bits) |
| 2 | MSR | Cyan Remaining (32 bits, hi/lo) |
| 3 | NMSR | |
| 4 | MSR | Magenta Remaining (32 bits, hi/lo) |
| 5 | NMSR | |
| 6 | MSR | Yellow Remaining (32 bits, hi/lo) |
| 7 | NMSR | |
| 8-11 | RO | For future expansion = 0 (64 bits) |
| 12-15 | RO | Random bit data (64 bits) |

Stage 2: Determine Keys for Systems and Consumables

Once the decision has been made as to which Systems and consumables are to share the same keys, those keys must be defined. The values for $K_1$ and $K_2$ must therefore be determined. In most cases, $K_1$ and $K_2$ will be generated once for all time. All Systems and consumables that have to work together (both now and in the future) need to have the same $K_1$ and $K_2$ values. $K_1$ and $K_2$ must therefore be kept secret since the entire security mechanism for the System/Consumable combination is made void if the keys are compromised. If the keys are compromised, the damage depends on the number of systems and consumables, and the ease to which they can be reprogrammed with new non-compromised keys: In the case of a photocopier with toner cartridges, the worst case is that a clone manufacturer could then manufacture their own Authentication Chips (or worse, buy them), program the chips with the known keys, and then insert them into their own consumables. In the case of a car with car-keys, each car has a different set of keys. This leads to two possible general scenarios. The first is that after the car and car-keys are programmed with the keys, $K_1$ and $K_2$ are deleted so no record of their values are kept, meaning that there is no way to compromise $K_1$ and $K_2$. However no more car-keys can be made for that car without reprogramming the car's Authentication Chip. The second scenario is that the car manufacturer keeps $K_1$ and $K_2$, and new keys can be made for the car. A compromise of $K_1$ and $K_2$ means that someone could make a car-key specifically for a particular car.

The keys and random data used in the Authentication Chips must therefore be generated by a means that is non-deterministic (a completely computer generated pseudo-random number cannot be used because it is deterministic – knowledge of the generator's seed gives all future numbers). $K_1$ and $K_2$ should be generated by a physically random process, and not by a computer. However, random bit generators based on natural sources of randomness are subject to influence by external factors and also to malfunction. It is imperative that such devices be tested periodically for statistical randomness.

A simple yet useful source of random numbers is the **Lavarand** ® system from SGI. This generator uses a digital camera to photograph six lava lamps every few minutes. Lava lamps contain chaotic turbulent systems. The resultant digital images are fed into an SHA-1 implementation that produces a 7-way hash, resulting in a 160-bit value from

every 7th bye from the digitized image. These 7 sets of 160 bits total 140 bytes. The 140 byte value is fed into a BBS generator to position the start of the output bitstream. The output 160 bits from the BBS would be the key or the Authentication chip 53.

An extreme example of a non-deterministic random process is someone flipping a coin 160 times for $K_1$ and 160 times for $K_2$ in a clean room. With each head or tail, a 1 or 0 is entered on a panel of a Key Programmer Device. The process must be undertaken with several observers (for verification) in silence (someone may have a hidden microphone). The point to be made is that secure data entry and storage is not as simple as it sounds. The physical security of the Key Programmer Device and accompanying Programming Station requires an entire document of its own. Once keys $K_1$ and $K_2$ have been determined, they must be kept for as long as Authentication Chips need to be made that use the key. In the first car/car-key scenario $K_1$ and $K_2$ are destroyed after a single System chip and a few consumable chips have been programmed. In the case of the photocopier / toner cartridge, $K_1$ and $K_2$ must be retained for as long as the toner-cartridges are being made for the photocopiers. The keys must be kept securely.

Stage 3: Determine MinTicks for Systems and Consumables

The value of MinTicks depends on the operating clock speed of the Authentication Chip (System specific) and the notion of what constitutes a reasonable time between RD or TST function calls (application specific). The duration of a single tick depends on the operating clock speed. This is the maximum of the input clock speed and the Authentication Chip's clock-limiting hardware. For example, the Authentication Chip's clock-limiting hardware may be set at 10 MHz (it is not changeable), but the input clock is 1 MHz. In this case, the value of 1 tick is based on 1 MHz, not 10 MHz. If the input clock was 20 MHz instead of 1 MHz, the value of 1 tick is based on 10 MHz (since the clock speed is limited to 10 MHz). Once the duration of a tick is known, the MinTicks value can be set. The value for MinTicks is the minimum number of ticks required to pass between calls to RD or RND key-based functions. Suppose the input clock speed matches the maximum clock speed of 10 MHz. If we want a minimum of 1 second between calls to TST, the value for MinTicks is set to 10,000,000. Even a value such as 2 seconds might be a completely reasonable value for a System such as a printer (one authentication per page, and one page produced every 2 or 3 seconds).

Stage 4: Program Keys, Random Seed, MinTicks and Unused M

Authentication Chips are in an unknown state after manufacture. Alternatively, they have already been used in one consumable, and must be reprogrammed for use in another. Each Authentication Chip must be cleared and programmed with new keys and new state data. Clearing and subsequent programming of Authentication Chips must take place in a secure Programming Station environment.

Programming a Trusted System Authentication Chip

If the chip is to be a trusted System chip, a seed value for R must be generated. It must be a random number derived from a physically random process, and must not be 0. The following tasks must be undertaken, in the following order, and in a secure programming environment:

RESET the chip

CLR[]

Load R (160 bit register) with physically random data

SSI[$K_1$, $K_2$, R]

SMT[$MinTicks_{System}$]

The Authentication Chip is now ready for insertion into a System. It has been completely programmed. If the System Authentication Chips are stolen at this point, a clone manufacturer could use them to generate R, $F_{K1}[R]$ pairs in order to launch a known text attack on $K_1$, or to use for launching a partially chosen-text attack on $K_2$. This is no different to the purchase of a number of Systems, each containing a trusted Authentication Chip. The security relies on the strength of the Authentication protocols and the randomness of $K_1$ and $K_2$.

Programming a Non-Trusted Consumable Authentication Chip

If the chip is to be a non-trusted Consumable Authentication Chip, the programming is slightly different to that of the trusted System Authentication Chip. Firstly, the seed value for R must be 0. It must have additional programming for M and the AccessMode values. The future use M[n] must be programmed with 0, and the random M[n] must be programmed with random data. The following tasks must be undertaken, in the following order, and in a secure programming environment:

RESET the chip
CLR[]
Load R (160 bit register) with 0
SSI[$K_1$, $K_2$, R]
Load X (256 bit register) with 0
Set bits in X corresponding to appropriate M[n] with physically random data
WR[X]
Load Y (32 bit register) with 0
Set bits in Y corresponding to appropriate M[n] with Read Only Access Modes
SAM[Y]
SMT[$MinTicks_{Consumable}$]

The non-trusted consumable chip is now ready to be programmed with the general state data. If the Authentication Chips are stolen at this point, an attacker could perform a limited chosen text attack. In the best situation, parts of M are Read Only (0 and random data), with the remainder of M completely chosen by an attacker (via the WR command). A number of RD calls by an attacker obtains $F_{K2}[M|R]$ for a limited M. In the worst situation, M can be completely chosen by an attacker (since all 256 bits are used for state data). In both cases however, the attacker cannot choose any value for R since it is supplied by calls to RND from a System Authentication Chip. The only way to obtain a chosen R is by a Brute Force attack. It should be noted that if Stages 4 and 5 are carried out on the same Programming Station (the preferred and ideal situation), Authentication Chips cannot be removed in between the stages. Hence there is no possibility of the Authentication Chips being stolen at this point. The decision to program the Authentication Chips at one or two times depends on the requirements of the System/Consumable manufacturer.

Stage 5: Program State Data and Access Modes

This stage is only required for consumable Authentication Chips, since M and AccessMode registers cannot be altered on System Authentication Chips. The future use and random values of M[n] have already been programmed in Stage 4. The remaining state data values need to be programmed and the associated Access Mode values need to be set. Bear in mind that the speed of this stage will be limited by the value stored in the MinTicks register. This stage is separated

from Stage 4 on account of the differences either in physical location or in time between where/when Stage 4 is performed, and where/when Stage 5 is performed. Ideally, Stages 4 and 5 are performed at the same time in the same Programming Station. Stage 4 produces valid Authentication Chips, but does not load them with initial state values (other than 0). This is to allow the programming of the chips to coincide with production line runs of consumables. Although Stage 5 can be run multiple times, each time setting a different state data value and Access Mode value, it is more likely to be run a single time, setting all the remaining state data values and setting all the remaining Access Mode values. For example, a production line can be set up where the batch number and serial number of the Authentication Chip is produced according to the physical consumable being produced. This is much harder to match if the state data is loaded at a physically different factory.

The Stage 5 process involves first checking to ensure the chip is a valid consumable chip, which includes a RD to gather the data from the Authentication Chip, followed by a WR of the initial data values, and then a SAM to permanently set the new data values. The steps are outlined here:

IsTrusted = GIT[]

If (IsTrusted), exit with error (wrong kind of chip!)

Call RND on a valid System chip to get a valid input pair

Call RD on chip to be programmed, passing in valid input pair

Load X (256 bit register) with results from a RD of Authentication Chip

Call TST on valid System chip to ensure X and consumable chip are valid

If (TST returns 0), exit with error (wrong consumable chip for system)

Set bits of X to initial state values

WR[X]

Load Y (32 bit register) with 0

Set bits of Y corresponding to Access Modes for new state values

SAM[Y]

Of course the validation (Steps 1 to 7) does not have to occur if Stage 4 and 5 follow on from one another on the same Programming Station. But it should occur in all other situations where Stage 5 is run as a separate programming process from Stage 4. If these Authentication Chips are now stolen, they are already programmed for use in a particular consumable. An attacker could place the stolen chips into a clone consumable. Such a theft would limit the number of cloned products to the number of chips stolen. A single theft should not create a supply constant enough to provide clone manufacturers with a cost-effective business. The alternative use for the chips is to save the attacker from purchasing the same number of consumables, each with an Authentication Chip, in order to launch a partially chosen text attack or brute force attack. There is no special security breach of the keys if such an attack were to occur.

## Manufacture

The circuitry of the Authentication Chip must be resistant to physical attack. A summary of manufacturing implementation guidelines is presented, followed by specification of the chip's physical defenses (ordered by attack).

### Guidelines for Manufacturing

The following are general guidelines for implementation of an Authentication Chip in terms of manufacture:

Standard process

Minimum size (if possible)

Clock Filter

Noise Generator

Tamper Prevention and Detection circuitry

Protected memory with tamper detection

Boot circuitry for loading program code

Special implementation of FETs for key data paths

Data connections in polysilicon layers where possible

OverUnderPower Detection Unit

No test circuitry

Standard Process

The Authentication Chip should be implemented with a standard manufacturing process (such as Flash). This is necessary to:

Allow a great range of manufacturing location options

Take advantage of well-defined and well-known technology

Reduce cost

Note that the standard process still allows physical protection mechanisms.

Minimum size

The Authentication chip 53 must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables. It is therefore desirable to keep the chip size as low as reasonably possible. Each Authentication Chip requires 802 bits of non-volatile memory. In addition, the storage required for optimized HMAC-SHA1 is 1024 bits. The remainder of the chip (state machine, processor, CPU or whatever is chosen to implement Protocol 3) must be kept to a minimum in order that the number of transistors is minimized and thus the cost per chip is minimized. The circuit areas that process the secret key information or could reveal information about the key should also be minimized (see Non-Flashing CMOS below for special data paths).

Clock Filter

The Authentication Chip circuitry is designed to operate within a specific clock speed range. Since the user directly supplies the clock signal, it is possible for an attacker to attempt to introduce race-conditions in the circuitry at specific times during processing. An example of this is where a high clock speed (higher than the circuitry is designed for) may prevent an XOR from working properly, and of the two inputs, the first may always be returned. These styles of transient fault attacks can be very efficient at recovering secret key information. The lesson to be learned from this is that the input clock signal cannot be trusted. Since the input clock signal cannot be trusted, it must be limited to operate up to a maximum frequency. This can be achieved a number of ways. One way to filter the clock signal is to use an edge detect unit passing the edge on to a delay, which in turn enables the input clock signal to pass through. Fig. 174 shows clock signal flow within the Clock Filter. The delay should be set so that the maximum clock speed is a particular frequency (e.g. about 4 MHz). Note that this delay is not programmable – it is fixed. The filtered clock signal would be further divided internally as required.

Noise Generator

Each Authentication Chip should contain a noise generator that generates continuous circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and add noise to the $I_{dd}$ signal.

Placement of the noise generator is not an issue on an Authentication Chip due to the length of the emission wavelengths. The noise generator is used to generate electronic noise, multiple state changes each clock cycle, and as a source of pseudo-random bits for the Tamper Prevention and Detection circuitry. A simple implementation of a noise generator is a 64-bit LFSR seeded with a non-zero number. The clock used for the noise generator should be running at the maximum clock rate for the chip in order to generate as much noise as possible.

Tamper Prevention and Detection circuitry

A set of circuits is required to test for and prevent physical attacks on the Authentication Chip. However what is actually detected as an attack may not be an intentional physical attack. It is therefore important to distinguish between these two types of attacks in an Authentication Chip:

where you can be certain that a physical attack has occurred.

where you cannot be certain that a physical attack has occurred.

The two types of detection differ in what is performed as a result of the detection. In the first case, where the circuitry can be certain that a true physical attack has occurred, erasure of Flash memory key information is a sensible action. In the second case, where the circuitry cannot be sure if an attack has occurred, there is still certainly something wrong. Action must be taken, but the action should not be the erasure of secret key information. A suitable action to take in the second case is a chip RESET. If what was detected was an attack that has permanently damaged the chip, the same conditions will occur next time and the chip will RESET again. If, on the other hand, what was detected was part of the normal operating environment of the chip, a RESET will not harm the key.

A good example of an event that circuitry cannot have knowledge about, is a power glitch. The glitch may be an intentional attack, attempting to reveal information about the key. It may, however, be the result of a faulty connection, or simply the start of a power-down sequence. It is therefore best to only RESET the chip, and not erase the key. If the chip was powering down, nothing is lost. If the System is faulty, repeated RESETs will cause the consumer to get the System repaired. In both cases the consumable is still intact.A good example of an event that circuitry can have knowledge about, is the cutting of a data line within the chip. If this attack is somehow detected, it could only be a result of a faulty chip (manufacturing defect) or an attack. In either case, the erasure of the secret information is a sensible step to take.

Consequently each Authentication Chip should have 2 Tamper Detection Lines as illustrated in Fig. – one for definite attacks, and one for possible attacks. Connected to these Tamper Detection Lines would be a number of Tamper Detection test units, each testing for different forms of tampering. In addition, we want to ensure that the Tamper Detection Lines and Circuits themselves cannot also be tampered with.

At one end of the Tamper Detection Line is a source of pseudo-random bits (clocking at high speed compared to the general operating circuitry). The Noise Generator circuit described above is an adequate source. The generated bits pass through two different paths – one carries the original data, and the other carries the inverse of the data. The wires carrying these bits are in the layer above the general chip circuitry (for example, the memory, the key manipulation circuitry etc). The wires must also cover the random bit generator. The bits are recombined at a number of places via an XOR gate. If the bits are different (they should be), a 1 is output, and used by the particular unit (for example, each output bit from a memory read should be ANDed with this bit value). The lines finally come together at the Flash memory Erase circuit, where a complete erasure is triggered by a 0 from the XOR. Attached to the line is a number of triggers, each detecting a physical attack on the chip. Each trigger has an oversize nMOS transistor attached to GND.

The Tamper Detection Line physically goes through this nMOS transistor. If the test fails, the trigger causes the Tamper Detect Line to become 0. The XOR test will therefore fail on either this clock cycle or the next one (on average), thus RESETing or erasing the chip. Fig. 175 illustrates the basic principle of a Tamper Detection Line in terms of tests and the XOR connected to either the Erase or RESET circuitry.

The Tamper Detection Line must go through the drain of an output transistor for each test, as illustrated by the oversize nMOS transistor layout of Fig. 176. :It is not possible to break the Tamper Detect Line since this would stop the flow of 1s and 0s from the random source. The XOR tests would therefore fail. As the Tamper Detect Line physically passes through each test, it is not possible to eliminate any particular test without breaking the Tamper Detect Line. It is important that the XORs take values from a variety of places along the Tamper Detect Lines in order to reduce the chances of an attack. Fig. 177 illustrates the taking of multiple XORs from the Tamper Detect Line to be used in the different parts of the chip. Each of these XORs can be considered to be generating a ChipOK bit that can be used within each unit or sub-unit.

A sample usage would be to have an OK bit in each unit that is ANDed with a given ChipOK bit each cycle. The OK bit is loaded with 1 on a RESET. If OK is 0, that unit will fail until the next RESET. If the Tamper Detect Line is functioning correctly, the chip will either RESET or erase all key information. If the RESET or erase circuitry has been destroyed, then this unit will not function, thus thwarting an attacker. The destination of the RESET and Erase line and associated circuitry is very context sensitive. It needs to be protected in much the same way as the individual tamper tests. There is no point generating a RESET pulse if the attacker can simply cut the wire leading to the RESET circuitry. The actual implementation will depend very much on what is to be cleared at RESET, and how those items are cleared. Finally, Fig. 178 shows how the Tamper Lines cover the noise generator circuitry of the chip. The generator and NOT gate are on one level, while the Tamper Detect Lines run on a level above the generator.

Protected memory with tamper detection

It is not enough to simply store secret information or program code in Flash memory. The Flash memory and RAM must be protected from an attacker who would attempt to modify (or set) a particular bit of program code or key information. The mechanism used must conform to being used in the Tamper Detection Circuitry (described above). The first part of the solution is to ensure that the Tamper Detection Line passes directly above each Flash or RAM bit. This ensures that an attacker cannot probe the contents of Flash or RAM. A breach of the covering wire is a break in the Tamper Detection Line. The breach causes the Erase signal to be set, thus deleting any contents of the memory. The high frequency noise on the Tamper Detection Line also obscures passive observation.

The second part of the solution for Flash is to use multi-level data storage, but only to use a subset of those multiple levels for valid bit representations. Normally, when multi-level Flash storage is used, a single floating gate holds more than one bit. For example, a 4-voltage-state transistor can represent two bits. Assuming a minimum and maximum voltage representing 00 and 11 respectively, the two middle voltages represent 01 and 10. In the Authentication Chip, we can use the two middle voltages to represent a single bit, and consider the two extremes to be invalid states. If an attacker attempts to force the state of a bit one way or the other by closing or cutting the gate's circuit, an invalid voltage (and hence invalid state) results.

The second part of the solution for RAM is to use a parity bit. The data part of the register can be checked against the parity bit (which will not match after an attack). The bits coming from Flash and RAM can therefore be validated by a number of test units (one per bit) connected to the common Tamper Detection Line. The Tamper Detection circuitry

would be the first circuitry the data passes through (thus stopping an attacker from cutting the data lines).

Boot circuitry for loading program code

Program code should be kept in multi-level Flash instead of ROM, since ROM is subject to being altered in a non-testable way. A boot mechanism is therefore required to load the program code into Flash memory (Flash memory is in an indeterminate state after manufacture). The boot circuitry must not be in ROM – a small state-machine would suffice. Otherwise the boot code could be modified in an undetectable way. The boot circuitry must erase all Flash memory, check to ensure the erasure worked, and then load the program code. Flash memory must be erased before loading the program code. Otherwise an attacker could put the chip into the boot state, and then load program code that simply extracted the existing keys. The state machine must also check to ensure that all Flash memory has been cleared (to ensure that an attacker has not cut the Erase line) before loading the new program code. The loading of program code must be undertaken by the secure Programming Station before secret information (such as keys) can be loaded.

Special implementation of FETs for key data paths

The normal situation for FET implementation for the case of a CMOS Inverter (which involves a pMOS transistor combined with an nMOS transistor) is shown in Fig. 179. During the transition, there is a small period of time where both the nMOS transistor and the pMOS transistor have an intermediate resistance. The resultant power-ground short circuit causes a temporary increase in the current, and in fact accounts for the majority of current consumed by a CMOS device. A small amount of infrared light is emitted during the short circuit, and can be viewed through the silicon substrate (silicon is transparent to infrared light). A small amount of light is also emitted during the charging and discharging of the transistor gate capacitance and transmission line capacitance.

For circuitry that manipulates secret key information, such information must be kept hidden. An alternative non-flashing CMOS implementation should therefore be used for all data paths that manipulate the key or a partially calculated value that is based on the key. The use of two non-overlapping clocks $\phi 1$ and $\phi 2$ can provide a non-flashing mechanism. $\phi 1$ is connected to a second gate of all nMOS transistors, and $\phi 2$ is connected to a second gate of all pMOS transistors. The transition can only take place in combination with the clock. Since $\phi 1$ and $\phi 2$ are non-overlapping, the pMOS and nMOS transistors will not have a simultaneous intermediate resistance. The setup is shown in Fig. 180.

Finally, regular CMOS inverters can be positioned near critical non-Flashing CMOS components. These inverters should take their input signal from the Tamper Detection Line above. Since the Tamper Detection Line operates multiple times faster than the regular operating circuitry, the net effect will be a high rate of light-bursts next to each non-Flashing CMOS component. Since a bright light overwhelms observation of a nearby faint light, an observer will not be able to detect what switching operations are occurring in the chip proper. These regular CMOS inverters will also effectively increase the amount of circuit noise, reducing the SNR and obscuring useful EMI.

There are a number of side effects due to the use of non-Flashing CMOS:

The effective speed of the chip is reduced by twice the rise time of the clock per clock cycle. This is not a problem for an Authentication Chip.

The amount of current drawn by the non-Flashing CMOS is reduced (since the short circuits do not occur). However, this is offset by the use of regular CMOS inverters.

Routing of the clocks increases chip area, especially since multiple versions of $\phi 1$ and $\phi 2$ are required to cater for

different levels of propagation. The estimation of chip area is double that of a regular implementation.

Design of the non-Flashing areas of the Authentication Chip are slightly more complex than to do the same with a with a regular CMOS design. In particular, standard cell components cannot be used, making these areas full custom. This is not a problem for something as small as an Authentication Chip, particularly when the entire chip does not have to be protected in this manner.

Connections in polysilicon layers where possible

Wherever possible, the connections along which the key or secret data flows, should be made in the polysilicon layers. Where necessary, they can be in metal 1, but must **never** be in the top metal layer (containing the Tamper Detection Lines).

OverUnderPower Detection Unit

Each Authentication Chip requires an OverUnderPower Detection Unit to prevent Power Supply Attacks. An OverUnderPower Detection Unit detects power glitches and tests the power level against a Voltage Reference to ensure it is within a certain tolerance. The Unit contains a single Voltage Reference and two comparators. The OverUnderPower Detection Unit would be connected into the RESET Tamper Detection Line, thus causing a RESET when triggered. A side effect of the OverUnderPower Detection Unit is that as the voltage drops during a power-down, a RESET is triggered, thus erasing any work registers.

No Test Circuitry

Test hardware on an Authentication Chip could very easily introduce vulnerabilities. As a result, the Authentication Chip should not contain any BIST or scan paths. The Authentication Chip must therefore be testable with external test vectors. This should be possible since the Authentication Chip is not complex.

Reading ROM

This attack depends on the key being stored in an addressable ROM. Since each Authentication Chip stores its authentication keys in internal Flash memory and not in an addressable ROM, this attack is irrelevant.

Reverse Engineering the Chip

Reverse engineering a chip is only useful when the security of authentication lies in the algorithm alone. However our Authentication Chips rely on a secret key, and not in the secrecy of the algorithm. Our authentication algorithm is, by contrast, public, and in any case, an attacker of a high volume consumable is assumed to have been able to obtain detailed plans of the internals of the chip. In light of these factors, reverse engineering the chip itself, as opposed to the stored data, poses no threat.

Usurping the Authentication Process

There are several forms this attack can take, each with varying degrees of success. In all cases, it is assumed that a clone manufacturer will have access to both the System and the consumable designs. An attacker may attempt to build a chip that tricks the System into returning a valid code instead of generating an authentication code. This attack is not possible for two reasons. The first reason is that System Authentication chips and Consumable Authentication Chips, although physically identical, are programmed differently. In particular, the RD opcode and the RND opcode are the same, as are the WR and TST opcodes. A System authentication Chip cannot perform a RD command since every call is interpreted as a call to RND instead. The second reason this attack would fail is that separate serial data lines are provided from the System to the System and Consumable Authentication Chips. Consequently neither chip can see what is being transmitted to or received from the other. If the attacker builds a clone chip that ignores WR commands

(which decrement the consumable remaining), Protocol 3 ensures that the subsequent RD will detect that the WR did not occur. The System will therefore not go ahead with the use of the consumable, thus thwarting the attacker. The same is true if an attacker simulates loss of contact before authentication – since the authentication does not take place, the use of the consumable doesn't occur. An attacker is therefore limited to modifying each System in order for clone consumables to be accepted

Modification of System

The simplest method of modification is to replace the System's Authentication Chip with one that simply reports success for each call to TST. This can be thwarted by System calling TST several times for each authentication, with the first few times providing false values, and expecting a fail from TST. The final call to TST would be expected to succeed. The number of false calls to TST could be determined by some part of the returned result from RD or from the system clock. Unfortunately an attacker could simply rewire System so that the new System clone authentication chip 53 can monitor the returned result from the consumable chip or clock. The clone System Authentication Chip would only return success when that monitored value is presented to its TST function. Clone consumables could then return any value as the hash result for RD, as the clone System chip would declare that value valid. There is therefore no point for the System to call the System Authentication Chip multiple times, since a rewiring attack will only work for the System that has been rewired, and not for all Systems. A similar form of attack on a System is a replacement of the System ROM. The ROM program code can be altered so that the Authentication never occurs. There is nothing that can be done about this, since the System remains in the hands of a consumer. Of course this would void any warranty, but the consumer may consider the alteration worthwhile if the clone consumable were extremely cheap and more readily available than the original item.

The System/consumable manufacturer must therefore determine how likely an attack of this nature is. Such a study must include given the pricing structure of Systems and Consumables, frequency of System service, advantage to the consumer of having a physical modification performed, and where consumers would go to get the modification performed. The limit case of modifying a system is for a clone manufacturer to provide a completely clone System which takes clone consumables. This may be simple competition or violation of patents. Either way, it is beyond the scope of the Authentication Chip and depends on the technology or service being cloned.

Direct viewing of chip operation by conventional probing

In order to view the chip operation, the chip must be operating. However, the Tamper Prevention and Detection circuitry covers those sections of the chip that process or hold the key. It is not possible to view those sections through the Tamper Prevention lines. An attacker cannot simply slice the chip past the Tamper Prevention layer, for this will break the Tamper Detection Lines and cause an erasure of all keys at power-up. Simply destroying the erasure circuitry is not sufficient, since the multiple ChipOK bits (now all 0) feeding into multiple units within the Authentication Chip will cause the chip's regular operating circuitry to stop functioning. To set up the chip for an attack, then, requires the attacker to delete the Tamper Detection lines, stop the Erasure of Flash memory, and somehow rewire the components that relied on the ChipOK lines. Even if all this could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

Direct viewing of the non-volatile memory

If the Authentication Chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the keys could probably be viewed directly using an STM or SKM. However, slicing the chip

to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling, or chemical mechanical polishing will almost certainly discharge the small charges present on the floating gates. This is true of regular Flash memory, but even more so of multi-level Flash memory.

Viewing the light bursts caused by state changes

All sections of circuitry that manipulate secret key information are implemented in the non-Flashing CMOS described above. This prevents the emission of the majority of light bursts. Regular CMOS inverters placed in close proximity to the non-Flashing CMOS will hide any faint emissions caused by capacitor charge and discharge. The inverters are connected to the Tamper Detection circuitry, so they change state many times (at the high clock rate) for each non-Flashing CMOS state change.

Monitoring EMI

The Noise Generator described above will cause circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and thus obscure any meaningful reading of internal data transfers.

Viewing $I_{dd}$ fluctuations

The solution against this kind of attack is to decrease the SNR in the $I_{dd}$ signal. This is accomplished by increasing the amount of circuit noise and decreasing the amount of signal. The Noise Generator circuit (which also acts as a defense against EMI attacks) will also cause enough state changes each cycle to obscure any meaningful information in the $I_{dd}$ signal. In addition, the special Non-Flashing CMOS implementation of the key-carrying data paths of the chip prevents current from flowing when state changes occur. This has the benefit of reducing the amount of signal.

Differential Fault Analysis

Differential fault bit errors are introduced in a non-targeted fashion by ionization, microwave radiation, and environmental stress. The most likely effect of an attack of this nature is a change in Flash memory (causing an invalid state) or RAM (bad parity). Invalid states and bad parity are detected by the Tamper Detection Circuitry, and cause an erasure of the key. Since the Tamper Detection Lines cover the key manipulation circuitry, any error introduced in the key manipulation circuitry will be mirrored by an error in a Tamper Detection Line. If the Tamper Detection Line is affected, the chip will either continually RESET or simply erase the key upon a power-up, rendering the attack fruitless. Rather than relying on a non-targeted attack and hoping that "just the right part of the chip is affected in just the right way", an attacker is better off trying to introduce a targeted fault (such as overwrite attacks, gate destruction etc). For information on these targeted fault attacks, see the relevant sections below.

Clock Glitch Attacks

The Clock Filter (described above) eliminates the possibility of clock glitch attacks.

Power Supply Attacks

The OverUnderPower Detection Unit (described above) eliminates the possibility of power supply attacks.

Overwriting ROM

Authentication Chips store Program code, keys and secret information in Flash memory, and not in ROM. This attack is therefore not possible.

Modifying EEPROM/Flash

Authentication Chips store Program code, keys and secret information in Flash memory. However, Flash memory is covered by two Tamper Prevention and Detection Lines. If either of these lines is broken (in the process of destroying a gate) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from

Flash memory. However, even if the attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the Authentication Chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash, detection circuitry will cause the Erasure Tamper Detection Line to be triggered – thereby erasing the remainder of Flash memory and RESETing the chip. A Modify EEPROM/Flash Attack is therefore fruitless.

Gate Destruction Attacks

Gate Destruction Attacks rely on the ability of an attacker to modify a single gate to cause the chip to reveal information during operation. However any circuitry that manipulates secret information is covered by one of the two Tamper Prevention and Detection lines. If either of these lines is broken (in the process of destroying a gate) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from Flash memory. To launch this kind of attack, an attacker must first reverse-engineer the chip to determine which gate(s) should be targeted. Once the location of the target gates has been determined, the attacker must break the covering Tamper Detection line, stop the Erasure of Flash memory, and somehow rewire the components that rely on the ChipOK lines. Rewiring the circuitry cannot be done without slicing the chip, and even if it could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

Overwrite Attacks

An Overwrite Attack relies on being able to set individual bits of the key without knowing the previous value. It relies on probing the chip, as in the Conventional Probing Attack and destroying gates as in the Gate Destruction Attack. Both of these attacks (as explained in their respective sections), will not succeed due to the use of the Tamper Prevention and Detection Circuitry and ChipOK lines. However, even if the attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the Authentication Chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash detection circuitry will cause the Erasure Tamper Detection Line to be triggered – thereby erasing the remainder of Flash memory and RESETing the chip. In the same way, a parity check on tampered values read from RAM will cause the Erasure Tamper Detection Line to be triggered. An Overwrite Attack is therefore fruitless.

Memory Remanence Attack

Any working registers or RAM within the Authentication Chip may be holding part of the authentication keys when power is removed. The working registers and RAM would continue to hold the information for some time after the removal of power. If the chip were sliced so that the gates of the registers/RAM were exposed, without discharging them, then the data could probably be viewed directly using an STM. The first defense can be found above, in the description of defense against Power Glitch Attacks. When power is removed, all registers and RAM are cleared, just as the RESET condition causes a clearing of memory. The chances then, are less for this attack to succeed than for a reading of the Flash memory. RAM charges (by nature) are more easily lost than Flash memory. The slicing of the chip to reveal the RAM will certainly cause the charges to be lost (if they haven't been lost simply due to the memory not being refreshed and the time taken to perform the slicing). This attack is therefore fruitless.

Chip Theft Attack

There are distinct phases in the lifetime of an Authentication Chip. Chips can be stolen when at any of these stages:

After manufacture, but before programming of key

After programming of key, but before programming of state data

After programming of state data, but before insertion into the consumable or system

After insertion into the system or consumable

A theft in between the chip manufacturer and programming station would only provide the clone manufacturer with blank chips. This merely compromises the sale of Authentication chips, not anything authenticated by the Authentication chips. Since the programming station is the only mechanism with consumable and system product keys, a clone manufacturer would not be able to program the chips with the correct key. Clone manufacturers would be able to program the blank chips for their own Systems and Consumables, but it would be difficult to place these items on the market without detection. The second form of theft can only happen in a situation where an Authentication Chip passes through two or more distinct programming phases. This is possible, but unlikely. In any case, the worst situation is where no state data has been programmed, so all of M is read/write. If this were the case, an attacker could attempt to launch an Adaptive Chosen Text Attack on the chip. The HMAC-SHA1 algorithm is resistant to such attacks. The third form of theft would have to take place in between the programming station and the installation factory. The Authentication chips would already be programmed for use in a particular system or for use in a particular consumable. The only use these chips have to a thief is to place them into a clone System or clone Consumable. Clone systems are irrelevant – a cloned System would not even require an authentication chip 53. For clone Consumables, such a theft would limit the number of cloned products to the number of chips stolen. A single theft should not create a supply constant enough to provide clone manufacturers with a cost-effective business.The final form of theft is where the System or Consumable itself is stolen. When the theft occurs at the manufacturer, physical security protocols must be enhanced. If the theft occurs anywhere else, it is a matter of concern only for the owner of the item and the police or insurance company. The security mechanisms that the Authentication Chip uses assume that the consumables and systems are in the hands of the public. Consequently, having them stolen makes no difference to the security of the keys.

Authentication Chip Design

The Authentication Chip has a physical and a logical external interface. The physical interface defines how the Authentication Chip can be connected to a physical System, and the logical interface determines how that System can communicate with the Authentication Chip.

Physical Interface

The Authentication Chip is a small 4-pin CMOS package (actual internal size is approximately 0.30 $mm^2$ using 0.25 µm Flash process). The 4 pins are GND, CLK, Power, and Data. Power is a nominal voltage. If the voltage deviates from this by more than a fixed amount, the chip will RESET. The recommended clock speed is 4-10 MHz. Internal circuitry filters the clock signal to ensure that a safe maximum clock speed is not exceeded. Data is transmitted and received one bit at a time along the serial data line. The chip performs a RESET upon power-up, power-down. In

addition, tamper detection and prevention circuitry in the chip will cause the chip to either RESET or erase Flash memory (depending on the attack detected) if an attack is detected. A special Programming Mode is enabled by holding the CLK voltage at a particular level. This is defined further in the next section.

Logical Interface

The Authentication Chip has two operating modes – a *Normal Mode* and a *Programming Mode*. The two modes are required because the operating program code is stored in Flash memory instead of ROM (for security reasons). The Programming mode is used for testing purposes after manufacture and to load up the operating program code, while the normal mode is used for all subsequent usage of the chip.

Programming Mode

The Programming Mode is enabled by holding a specific voltage on the CLK line for a given amount of time. When the chip enters Programming Mode, all Flash memory is erased (including all secret key information and any program code). The Authentication Chip then validates the erasure. If the erasure was successful, the Authentication Chip receives 384 bytes of data corresponding to the new program code. The bytes are transferred in order $byte_0$ to $byte_{383}$. The bits are transferred from $bit_0$ to $bit_7$. Once all 384 bytes of program code have been loaded, the Authentication Chip hangs. If the erasure was not successful, the Authentication Chip will hang without loading any data into the Flash memory. After the chip has been programmed, it can be restarted. When the chip is RESET with a normal voltage on the CLK line, Normal Mode is entered.

Normal Mode

Whenever the Authentication Chip is not in Programming Mode, it is in Normal Mode. When the Authentication Chip starts up in Normal Mode (for example a power-up RESET), it executes the program currently stored in the program code region of Flash memory. The program code implements a communication mechanism between the System and Authentication Chip, accepting commands and data from the System and producing output values. Since the Authentication Chip communicates serially, bits are transferred one at a time.The System communicates with the Authentication Chips via a simple operation command set. Each command is defined by 3-bit opcode. The interpretation of the opcode depends on the current value of the IsTrusted bit and the IsWritten bit.

The following operations are defined:

| Op | T | W | Mn | Input | Output | Description |
|---|---|---|---|---|---|---|
| 000 | - | - | CLR | - | - | Clear |
| 001 | 0 | 0 | SSI | [160, 160, 160] | - | Set Secret Information |
| 010 | 0 | 1 | RD | [160, 160] | [256, 160] | Read M securely |
| 010 | 1 | 1 | RND | - | [160, 160] | Random |
| 011 | 0 | 1 | WR | [256] | - | Write M |
| 011 | 1 | 1 | TST | [256, 160] | [1] | Test |
| 100 | 0 | 1 | SAM | [32] | [32] | Set Access Mode |
| 101 | - | 1 | GIT | - | [1] | Get Is Trusted |
| 110 | - | 1 | SMT | [32] | - | Set MinTicks |

**Op** = Opcode, **T** = IsTrusted value, **W** = IsWritten value,

**Mn** = Mnemonic, [n] = number of bits required for parameter

Any command not defined in this table is interpreted as NOP (**No operation**). Examples include opcodes 110 and 111 (regardless of IsTrusted or IsWritten values), and any opcode other than SSI when IsWritten = 0. Note that the opcodes for RD and RND are the same, as are the opcodes for WR and TST. The actual command run upon receipt of the opcode will depend on the current value of the IsTrusted bit (as long as IsWritten is 1). Where the IsTrusted bit is clear, RD and WR functions will be called. Where the IsTrusted bit is set, RND and TST functions will be called. The two sets of commands are mutually exclusive between trusted and non-trusted Authentication Chips. In order to execute a command on an Authentication Chip, a client (such as System) sends the command opcode followed by the required input parameters for that opcode. The opcode is sent least significant bit through to most significant bit. For example, to send the SSI command, the bits **1**, **0**, and **0** would be sent in that order. Each input parameter is sent in the same way, least significant bit first through to most significant bit last. Return values are read in the same way – least significant bit first and most significant bit last. The client must know how many bits to retrieve.

In some cases, the output bits from one chip's command can be fed directly as the input bits to another chip's command. An example of this is the RND and RD commands. The output bits from a call to RND on a trusted Authentication Chip do not have to be kept by System. Instead, System can transfer the output bits directly to the input of the non-trusted Authentication Chip's RD command. The description of each command points out where this is so.

Each of the commands is examined in detail in the subsequent sections. Note that some algorithms are specifically designed because the permanent registers are kept in Flash memory.

Registers

The memory within the Authentication Chip contains some non-volatile memory to store the variables required by the Authentication Protocol. The following non-volatile (Flash) variables are defined:

| Variable Name | Size (in bits) | Description |
|---|---|---|
| M[0..15] | 256 | 16 words (each 16 bits) containing state data such as serial numbers, media remaining etc. |
| $K_1$ | 160 | Key used to transform R during authentication. |
| $K_2$ | 160 | Key used to transform M during authentication. |
| R | 160 | Current random number |
| AccessMode[0..15] | 32 | The 16 sets of 2-bit AccessMode values for M[n]. |
| MinTicks | 32 | The minimum number of clock ticks between calls to key-based functions |
| SIWritten | 1 | If set, the secret key information ($K_1$, $K_2$, and R) has been written to the chip. If clear, the secret information has not been written yet. |
| IsTrusted | 1 | If set, the RND and TST functions can be called, but RD and WR functions cannot be called.<br>If clear, the RND and TST functions cannot be called, but RD and WR functions can be called. |
| Total bits | 802 | |

Architecture Overview

This section chapter provides the high-level definition of a purpose-built CPU capable of implementing the functionality required of an Authentication Chip. Note that this CPU is not a general purpose CPU. It is tailor-made for implementing the Authentication logic. The authentication commands that a user of an Authentication Chip sees, such as WRITE, TST, RND etc are all implemented as small programs written in the CPU instruction set. The CPU contains a 32-bit Accumulator (which is used in most operations), and a number of registers. The CPU operates on 8-bit instructions specifically tailored to implementing authentication logic. Each 8-bit instruction typically consists of a 4-bit opcode, and a 4-bit operand.

Operating Speed

An internal Clock Frequency Limiter Unit prevents the chip from operating at speeds any faster than a predetermined frequency. The frequency is built into the chip during manufacture, and cannot be changed. The frequency is recommended to be about 4-10 MHz.

Composition and Block Diagram

The Authentication Chip contains the following components:

| Unit Name | CMOS Type | Description |
|---|---|---|
| Clock Frequency Limiter | Normal | Ensures the operating frequency of the Authentication Chip does not exceed a specific maximum frequency. |
| OverUnderPower Detection Unit | Normal | Ensures that the power supply remains in a valid operating range. |
| Programming Mode Detection Unit | Normal | Allows users to enter Programming Mode. |
| Noise Generator | Normal | For generating $I_{dd}$ noise and for use in the Tamper Prevention and Detection circuitry. |
| State Machine | Normal | for controlling the two operating modes of the chip (Programming Mode and Normal Mode). This includes generating the two operating cycles of the CPU, stalling during long command operations, and storing the op-code and operand during operating cycles. |
| I/O Unit | Normal | Responsible for communicating serially with the outside world. |
| ALU | Non-flashing | Contains the 32-bit accumulator as well as the general mathematical and logical operators. |
| MinTicks Unit | Normal (99%), Non-flashing (1%) | Responsible for a programmable minimum delay (via a countdown) between certain key-based operations. |
| Address Generator Unit | Normal (99%), Non-flashing (1%) | Generates direct, indirect, and indexed addresses as required by specific operands. |
| Program Counter Unit | Normal | Includes the 9 bit PC (program counter), as well as logic for branching and subroutine control |
| Memory Unit | Non-flashing | Addressed by 9 bits of address. It contains an 8-bit wide program Flash memory, and 32-bit wide Flash memory, RAM, and look-up tables. Also contains Programming Mode circuitry to enable loading of program code. |

Fig. 181 illustrates a schematic block diagram of the Authentication Chip. The tamper prevention and Detection Circuitry is not shown: The Noise Generator, OverUnderPower Detection Unit, and ProgrammingMode Detection Unit are connected to the Tamper Prevention and Detection Circuitry and not to the remaining units.

Memory Map

Fig. 182 illustrates an example memory map. Although the Authentication Chip does not have external memory, it does have internal memory. The internal memory is addressed by 9 bits, and is either 32-bits wide or 8-bits wide

(depending on address). The 32-bit wide memory is used to hold the non-volatile data, the variables used for HMAC-SHA1, and constants. The 8-bit wide memory is used to hold the program and the various jump tables used by the program. The address breakup (including reserved memory ranges) is designed to optimize address generation and decoding.

Constants

Fig. 183 illustrates an example of the constants memory map. The Constants region consists of 32-bit constants. These are the simple constants (such as 32-bits of all 0 and 32-bits of all 1), the constants used by the HMAC algorithm, and the constants $y_{0-3}$ and $h_{0-4}$ required for use in the SHA-1 algorithm. None of these values are affected by a RESET. The only opcode that makes use of constants is LDK. In this case, the operands and the memory placement are closely linked, in order to minimize the address generation and decoding.

RAM

Fig. 184 illustrates an example of the RAM memory map. The RAM region consists of the 32 parity-checked 32-bit registers required for the general functioning of the Authentication Chip, *but only during the operation of the chip.* RAM is volatile memory, which means that once power is removed, the values are lost. Note that in actual fact, memory retains its value for some period of time after power-down (due to memory remnance), but cannot be considered to be available upon power-up. This has issues for security that are addressed in other sections of this document. RAM contains the variables used for the HMAC-SHA1 algorithm, namely: A-E, the temporary variable T, space for the 160-bit working hash value H, space for temporary storage of a hash result (required by HMAC) B160, and the space for the 512 bits of expanded hashing memory X. All RAM values are cleared to 0 upon a RESET, although any program code should not take this for granted. Opcodes that make use of RAM addresses are LD, ST, ADD, LOG, XOR, and RPL. In all cases, the operands and the memory placement are closely linked, in order to minimize the address generation and decoding (multiword variables are stored most significant word first).

Flash Memory – Variables

Fig. 185 illustrates an example of the Flash memory variables memory map. The Flash memory region contains the non-volatile information in the Authentication Chip. Flash memory retains its value after power is removed, and can be expected to be unchanged when the power is next turned on. The non-volatile information kept in multi-state Flash memory includes the two 160-bit keys ($K_1$ and $K_2$), the current random number value (R), the state data (M), the MinTicks value (MT), the AccessMode value (AM), and the IsWritten (ISW) and IsTrusted (IST) flags.Flash values are unchanged by a RESET, but are cleared (to 0) upon entering Programming Mode. Operations that make use of Flash addresses are LD, ST, ADD, RPL, ROR, CLR, and SET. In all cases, the operands and the memory placement are closely linked, in order to minimize the address generation and decoding. Multiword variables $K_1$, $K_2$, and M are stored most significant word first due to addressing requirements. The addressing scheme used is a base address offset by an index that starts at N and ends at 0. Thus $M_N$ is the first word accessed, and $M_0$ is the last 32-bit word accessed in loop processing. Multiword variable R is stored least significant word first for ease of LFSR generation using the same indexing scheme.

Flash Memory – Program

Fig. 186 illustrates an example of the Flash memory program memory map. The second multi-state Flash memory region is 384 x 8-bits. The region contains the address tables for the JSR, JSI and TBR instructions, the offsets for the DBR commands, constants and the program itself. The Flash memory is unaffected by a RESET, but is cleared (to 0)

upon entering Programming Mode. Once Programming Mode has been entered, the 8-bit Flash memory can be loaded with a new set of 384 bytes. Once this has been done, the chip can be RESET and the normal chip operations can occur.

Registers

A number of registers are defined in the Authentication Chip. They are used for temporary storage during function execution. Some are used for arithmetic functions, others are used for counting and indexing, and others are used for serial I/O. These registers do not need to be kept in non-volatile (Flash) memory. They can be read or written without the need for an erase cycle (unlike Flash memory). Temporary storage registers that contain secret information still need to be protected from physical attack by Tamper Prevention and Detection circuitry and parity checks.

All registers are cleared to 0 on a RESET. However, program code should not assume any particular state, and set up register values appropriately. Note that these registers do not include the various OK bits defined for the Tamper Prevention and Detection circuitry. The OK bits are scattered throughout the various units and are set to 1 upon a RESET.

Cycle

The 1-bit Cycle value determines whether the CPU is in a Fetch cycle (0) or an Execute cycle (1). Cycle is actually derived from a 1-bit register that holds the previous Cycle value. Cycle is not directly accessible from the instruction set. It is an internal register only.

Program Counter

A 6-level deep 9-bit Program Counter Array (PCA) is defined. It is indexed by a 3-bit Stack Pointer (SP). The current Program Counter (PC), containing the address of the currently executing instruction, is effectively PCA[SP]. In addition, a 9-bit Adr register is defined, containing the resolved address of the current memory reference (for indexed or indirect memory accesses). The PCA, SP, and Adr registers are not directly accessible from the instruction set. They are internal registers only

CMD

The 8-bit CMD register is used to hold the currently executing command. While the CMD register is not directly accessible from the instruction set, and is an internal register only.

Accumulator and Z flag

The Accumulator is a 32-bit general-purpose register. It is used as one of the inputs to all arithmetic operations, and is the register used for transferring information between memory registers. The Z register is a 1-bit flag, and is updated each time the Accumulator is written to. The Z register contains the zero-ness of the Accumulator. Z = 1 if the last value written to the Accumulator was 0, and 0 if the last value written was non-0. Both the Accumulator and Z registers are directly accessible from the instruction set.

Counters

A number of special purpose counters/index registers are defined:

| Name | Register Size | Bits | Description |
|---|---|---|---|
| C1 | 1 x 3 | 3 | Counter used to index arrays:<br>AE, B160, M, H, y, and h. |
| C2 | 1 x 5 | 5 | General purpose counter |
| $N_{1-4}$ | 4 x 4 | 16 | Used to index array X |

All these counter registers are directly accessible from the instruction set. Special instructions exist to load them with specific values, and other instructions exist to decrement or increment them, or to branch depending on the whether or not the specific counter is zero. There are also 2 special flags (not registers) associated with C1 and C2, and these flags hold the zero-ness of C1 or C2. The flags are used for loop control, and are listed here, for although they are not registers, they can be tested like registers.

| Name | Description |
|---|---|
| C1Z | 1 = C1 is current zero, 0 =C1 is currently non-zero. |
| C2Z | 1 = C2 is current zero, 0 =C2 is currently non-zero. |

Flags

A number of 1-bit flags, corresponding to CPU operating modes, are defined:

| Name | Bits | Description |
|---|---|---|
| WE | 1 | WriteEnable for X register array:<br>0 = Writes to X registers become no-ops<br>1 = Writes to X registers are carried out |
| K2MX | 1 | 0 = K1 is accessed during K references. Reads from M are interpreted as reads of 0<br>1 = K2 is accessed during K references. Reads from M succeed. |

All these 1-bit flags are directly accessible from the instruction set. Special instructions exist to set and clear these flags. Registers used for Write Integrity

| Name | Bits | Description |
|---|---|---|
| EE | 1 | Corresponds to the EqEncountered variable in the WR command pseudocode. Used during the writing of multi-precision data values to determine whether all more significant components have been equal to their previous values. |
| DE | 1 | Corresponds to the DecEncountered variable in the WR command pseudocode. Used during the writing of multi-precision data values to determine whether a more significant components has been decremented already. |

Registers used for I/O

Four 1-bit registers are defined for communication between the client (System) and the Authentication Chip. These registers are InBit, InBitValid, OutBit, and OutBitValid. InBit and InBitValid provide the means for clients to pass commands and data to the Authentication Chip. OutBit and OutBitValid provide the means for clients to get information from the Authentication Chip. A client sends commands and parameter bits to the Authentication Chip one bit at a time. Since the Authentication Chip is a slave device, from the Authentication Chip's point of view:

Reads from InBit will hang while InBitValid is clear. InBitValid will remain clear until the client has written the next input bit to InBit. Reading InBit clears the InBitValid bit to allow the next InBit to be read from the client. A client cannot write a bit to the Authentication Chip unless the InBitValid bit is clear.

Writes to OutBit will hang while OutBitValid is set. OutBitValid will remain set until the client has read the bit from OutBit. Writing OutBit sets the OutBitValid bit to allow the next OutBit to be read by the client. A client cannot read a bit from the Authentication Chip unless the OutBitValid bit is set.

Registers Used for Timing Access

A single 32-bit register is defined for use as a timer. The MTR (**MinTicksRemaining**) register decrements every time an instruction is executed. Once the MTR register gets to 0, it stays at zero. Associated with MTR is a 1-bit flag MTRZ, which contains the zero-ness of the MTR register. If MTRZ is 1, then the MTR register is zero. If MTRZ is 0, then the MTR register is not zero yet. MTR always starts off at the MinTicks value (after a RESET or a specific key-accessing function), and eventually decrements to 0. While MTR can be set and MTRZ tested by specific instructions, the value of MTR cannot be directly read by any instruction.

Register Summary

The following table summarizes all temporary registers (ordered by register name). It lists register names, size (in bits), as well as where the specified register can be found.

| Register Name | Bits | Parity | Where Found |
|---|---|---|---|
| Acc | 32 | 1 | Arithmetic Logic Unit |
| Adr | 9 | 1 | Address Generator Unit |
| AMT | 32 | | Arithmetic Logic Unit |
| C1 | 3 | 1 | Address Generator Unit |
| C2 | 5 | 1 | Address Generator Unit |
| CMD | 8 | 1 | State Machine |
| Cycle (Old = prev Cycle) | 1 | | State Machine |
| DE | 1 | | Arithmetic Logic Unit |
| EE | 1 | | Arithmetic Logic Unit |
| InBit | 1 | | Input Output Unit |
| InBitValid | 1 | | Input Output Unit |
| K2MX | 1 | | Address Generator Unit |
| MTR | 32 | 1 | MinTicks Unit |
| MTRZ | 1 | | MinTicks Unit |
| N[1-4] | 16 | 4 | Address Generator Unit |
| OutBit | 1 | | Input Output Unit |
| OutBitValid | 1 | | Input Output Unit |
| PCA | 54 | 6 | Program Counter Unit |
| RTMP | 1 | | Arithmetic Logic Unit |
| SP | 3 | 1 | Program Counter Unit |
| WE | 1 | | Memory Unit |
| Z | 1 | | Arithmetic Logic Unit |
| **Total bits** | **206** | **17** | |

Instruction Set

The CPU operates on 8-bit instructions specifically tailored to implementing authentication logic. The majority of 8-bit instruction consists of a 4-bit opcode, and a 4-bit operand. The high-order 4 bits contains the opcode, and the low-order 4 bits contains the operand.

Opcodes and Operands (Summary)

The opcodes are summarized in the following table:

| Opcode | Mnemonic | Simple Description |
|---|---|---|
| 0000 | TBR | Test and branch. |
| 0001 | DBR | Decrement and branch |
| 001 | JSR | Jump subroutine via table |
| 01000 | RTS | Return from subroutine |
| 01001 | JSI | Jump subroutine indirect |
| 0101 | SC | Set counter |
| 0110 | CLR | Clear specific flash registers |
| 0111 | SET | Set bits in specific flash register |
| 1000 | ADD | Add a 32 bit value to the Accumulator |
| 1001 | LOG | Logical operation (AND, and OR ) |
| 1010 | XOR | Exclusive-OR Accumulator with some value |
| 1011 | LD | Load Accumulator from specified location |
| 1100 | ROR | Rotate Accumulator right |
| 1101 | RPL | Replace bits |
| 1110 | LDK | Load Accumulator with a constant |
| 1111 | ST | Store Accumulator in specified location |

The following table is a summary of which operands can be used with which opcodes. The table is ordered alphabetically by opcode mnemonic. The binary value for each operand can be found in the subsequent tables.

| Opcode | Valid Operand |
|---|---|
| ADD | {A, B, C, D, E, T, MT, AM, AE[C1], B160[C1], H[C1], M[C1], K[C1], R[C1], X[N4]} |
| CLR | {WE, K2MX, M[C1], Group1, Group2} |
| DBR | {C1, C2}, Offset into DBR Table |
| JSI | {} |
| JSR | Offset into Table 1 |
| LD | {A, B, C, D, E, T, MT, AM, AE[C1], B160[C1], H[C1], M[C1], K[C1], R[C1], X[N4]} |
| LDK | {0x0000…, 0x3636…, 0x5C5C…, 0xFFFF, h[C1], y[C1]} |
| LOG | {AND, OR}, {A, B, C, D, E, T, MT, AM} |
| ROR | {InBit, OutBit, LFSR, RLFSR, IST, ISW, MTRZ, 1, 2, 27, 31} |
| RPL | {Init, MHI, MLO} |
| RTS | {} |
| SC | {C1, C2}, Offset into counter list |
| SET | {WE, K2MX, Nx, MTR, IST, ISW} |
| ST | {A, B, C, D, E, T, MT, AM, AE[C1], B160[C1], H[C1], M[C1], K[C1], R[C1], X[N4]} |
| TBR | {0, 1}, Offset into Table 1 |
| XOR | {A, B, C, D, E, T, MT, AM, X[N1], X[N2], X[N3], X[N4]} |

The following operand table shows the interpretation of the 4-bit operands where all 4 bits are used for direct interpretation.

| Operand | ADD, LD,ST | XOR | ROR | LDK | RPL | SET | CLR |
|---|---|---|---|---|---|---|---|
| 0000 | E | E | InBit | 0x00... | Init | WE | WE |
| 0001 | D | D | OutBit | 0x36... | - | K2MX | K2MX |
| 0010 | C | C | RB | 0x5C... | - | Nx | - |
| 0011 | B | B | XRB | 0xFF... | - | - | - |
| 0100 | A | A | IST | y[C1] | - | IST | - |
| 0101 | T | T | ISW | - | - | ISW | - |
| 0110 | MT | MT | MTRZ | - | - | MTR | - |
| 0111 | AM | AM | 1 | - | - | - | - |
| 1000 | AE[C1] | - | - | h[C1] | - | - | - |
| 1001 | B160[C1] | - | 2 | - | - | - | - |
| 1010 | H[C1] | - | 27 | - | - | - | - |
| 1011 | - | - | - | - | - | - | - |
| 1100 | R[C1] | X[N1] | 31 | - | - | - | R |
| 1101 | K[C1] | X[N2] | - | - | - | - | Group1 |
| 1110 | M[C1] | X[N3] | - | - | MLO | - | M[C1] |
| 1111 | X[N4] | X[N4] | - | - | MHI | - | Group2 |

The following instructions make a selection based upon the highest bit of the operand:

| $\text{Operand}_3$ | Which Counter? (DBR, SC) | Which operation? (LOG) | Which Value? (TBR) |
|---|---|---|---|
| 0 | C1 | AND | Zero |
| 1 | C2 | OR | Non-zero |

The lowest 3 bits of the operand are either offsets (DBR, TBR), values from a special table (SC) or as in the case of LOG, they select the second input for the logical operation. The interpretation matches the interpretation for the ADD, LD, and ST opcodes:

| Operand$_{2-0}$ | LOG Input2 | SC Value |
|---|---|---|
| 000 | E | 2 |
| 001 | D | 3 |
| 010 | C | 4 |
| 011 | B | 7 |
| 100 | A | 10 |
| 101 | T | 15 |
| 110 | MT | 19 |
| 111 | AM | 31 |

ADD – Add To Accumulator

**Mnemonic:** ADD

**Opcode:**1000

**Usage:** ADD Value

The ADD instruction adds the specified operand to the Accumulator via modulo $2^{32}$ addition. The operand is one of A, B, C, D, E, T, AM, MT, AE[C1], H[C1], B160[C1], R[C1], K[C1], M[C1], or X[N4]. The Z flag is also set during this operation, depending on whether the value loaded is zero or not.

CLR – Clear Bits

**Mnemonic:** CLR

**Opcode:**0110

**Usage:** CLR Flag/Register

The CLR instruction causes the specified internal flag or Flash memory registers to be cleared. In the case of Flash memory, although the CLR instruction takes some time the next instruction is stalled until the erasure of Flash memory has finished. The registers that can be cleared are WE and K2MX. The Flash memory that can be cleared are: R, M[C1], Group1, and Group2. Group1 is the IST and ISW flags. If these are cleared, then the only valid high level command is the SSI instruction. Group2 is the MT, AM, K1 and K2 registers. R is erased separately since it must be updated after each call to TST. M is also erased via an index mechanism to allow individual parts of M to be updated. There is also a corresponding SET instruction.

DBR – Decrement and Branch

**Mnemonic:** DBR

**Opcode:**0001

**Usage:** DBR Counter, Offset

This instruction provides the mechanism for building simple loops. The high hit of the operand selects between testing

C1 or C2 (the two counters). If the specified counter is non-zero, then the counter is decremented and the value at the given offset (sign extended) is added to the PC. If the specified counter is zero, it is decremented and processing continues at PC+1. The 8-entry offset table is stored at address 0 1100 0000 (the 64th entry of the program memory). The 8 bits of offset are treated as a signed number. Thus 0xFF is treated as –1, and 0x01 is treated as +1. Typically the value will be negative for use in loops.

JSI – Jump Subroutine Indirect

**Mnemonic:** JSI

**Opcode:**01001

**Usage:** JSI (Acc)

The JSI instruction allows the jumping to a subroutine dependant on the value currently in the Accumulator. The instruction pushes the current PC onto the stack, and loads the PC with a new value. The upper 8 bits of the new PC are loaded from Jump Table 2 (offset given by the lower 5 bits of the Accumulator), and the lowest bit of the PC is cleared to 0. Thus all subroutines must start at even addresses. The stack provides for 6 levels of execution (5 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the return value will be overwritten (since the stack wraps).

JSR – Jump Subroutine

**Mnemonic:** JSR

**Opcode:**001

**Usage:** JSR Offset

The JSR instruction provides for the most common usage of the subroutine construct. The instruction pushes the current PC onto the stack, and loads the PC with a new value. The upper 8 bits of the new PC value comes from Address Table 1, with the offset into the table provided by the 5-bit operand (32 possible addresses). The lowest bit of the new PC is cleared to 0. Thus all subroutines must start at even addresses. The stack provides for 6 levels of execution (5 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the return value will be overwritten (since the stack wraps).

LD – Load Accumulator

**Mnemonic:** LD

**Opcode:**1011

**Usage:** LD Value

The LD instruction loads the Accumulator from the specified operand. The operand is one of A, B, C, D, E, T, AM, MT, AE[C1], H[C1], B160[C1], R[C1], K[C1], M[C1], or X[N4]. The Z flag is also set during this operation, depending on whether the value loaded is zero or not.

LDK – Load Constant

**Mnemonic:** LDK

**Opcode:**1110

**Usage**: LDK Constant

The LDK instruction loads the Accumulator with the specified constant. The constants are those 32-bit values required for HMAC-SHA1 and all 0s and all 1s as most useful for general purpose processing. Consequently they are a choice of:

0x00000000

0x36363636

0x5C5C5C5C

0xFFFFFFFF

or from the h and y constant tables, indexed by C1. The h and y constant tables hold the 32-bit tabular constants required for HMAC-SHA1. The Z flag is also set during this operation, depending on whether the constant loaded is zero or not.

LOG – Logical Operation

**Mnemonic**: LOG

**Opcode**: 1001

**Usage**: LOG Operation Value

The LOG instruction performs 32-bit bitwise logical operations on the Accumulator and a specified value. The two operations supported by the LOG instruction are AND and OR. Bitwise NOT and XOR operations are supported by the XOR instruction. The 32-bit value to be ANDed or ORed with the accumulator is one of the following: A, B, C, D, E, T, MT and AM. The Z flag is also set during this operation, depending on whether resultant 32-bit value (loaded into the Accumulator) is zero or not.

ROR – Rotate Right

**Mnemonic**: ROR

**Opcode**: 1100

**Usage**: ROR Value

The ROR instruction provides a way of rotating the Accumulator right a set number of bits. The bit coming in at the top of the Accumulator (to become bit 31) can either come from the previous bit 0 of the Accumulator, or from an external 1-bit flag (such as a flag, or the serial input connection). The bit rotated out can also be output from the serial connection, or combined with an external flag. The allowed operands are: InBit, OutBit, LFSR, RLFSR, IST, ISW, MTRZ, 1, 2, 27, and 31. The Z flag is also set during this operation, depending on whether resultant 32-bit value (loaded into the Accumulator) is zero or not. In its simplest form, the operand for the ROR instruction is one of 1, 2, 27, 31, indicating how many bit positions the Accumulator should be rotated. For these operands, there is no external input or output – the bits of the Accumulator are merely rotated right. With operands IST, ISW, and MTRZ, the appropriate flag is transferred to the highest bit of the Accumulator. The remainder of the Accumulator is shifted right one bit position (bit31 becomes bit 30 etc), with lowest bit of the Accumulator shifted out. With operand InBit, the next serial input bit is transferred to the highest bit of the Accumulator. The InBitValid bit is then cleared. If there is no input bit available from the client yet, execution is suspended until there is one. The remainder of the Accumulator is shifted right one bit position (bit31 becomes bit 30 etc), with lowest bit of the Accumulator shifted out.

With operand OutBit, the Accumulator is shifted right one bit position. The bit shifted out from bit 0 is stored in the OutBit flag and the OutBitValid flag is set. It is therefore ready for a client to read. If the OutBitValid flag is already set, execution of the instruction stalls until the OutBit bit has been read by the client (and the OutBitValid flag cleared). The new bit shifted in to bit 31 should be considered garbage (actually the value currently in the InBit register). Finally, the RB and XRB operands allow the implementation of LFSRs and multiple precision shift registers. With RB, the bit shifted out (formally bit 0) is written to the RTMP register. The register currently in the RTMP register becomes the new bit 31 of the Accumulator. Performing multiple ROR RB commands over several 32-bit values implements a multiple precision rotate/shift right. The XRB operates in the same way as RB, in that the current value in the RTMP register becomes the new bit 31 of the Accumulator. However with the XRB instruction, the bit formally known as bit 0 does not simply replace RTMP (as in the RB instruction). Instead, it is XORed with RTMP, and the result stored in RTMP. This allows the implementation of long LFSRs, as required by the Authentication protocol.

RPL – Replace Bits

**Mnemonic**: RPL

**Opcode:**1101

**Usage**: ROR Value

The RPL instruction is designed for implementing the high level WRITE command in the Authentication Chip. The instruction is designed to replace the upper 16 bits of the Accumulator by the value that will eventually be written to the M array (dependant on the Access Mode value). The instruction takes 3 operands: Init, MHI, and MLO. The Init operand sets all internal flags and prepares the RPL unit within the ALU for subsequent processing. The Accumulator is transferred to an internal AccessMode register. The Accumulator should have been loaded from the AM Flash memory location before the call to RPL Init in the case of implementing the WRITE command, or with 0 in the case of implementing the TST command. The Accumulator is left unchanged. The MHI and MLO operands refer to whether the upper or lower 16 bits of M[C1] will be used in the comparison against the (always) upper 16 bits of the Accumulator. Each MHI and MLO instruction executed uses the subsequent 2 bits from the initialized AccessMode value. The first execution of MHI or MLO uses the lowest 2 bits, the next uses the second two bits etc.

RTS – Return From Subroutine

**Mnemonic**: RTS

**Opcode**:01000

**Usage**: RTS

The RTS instruction causes execution to resume at the instruction after the most recently executed JSR or JSI instruction. Hence the term: returning from the subroutine. In actuality, the instruction pulls the saved PC from the stack, adds 1, and resumes execution at the resultant address. Although 6 levels of execution are provided for (5 subroutines), it is the responsibility of the programmer to balance each JSR and JSI instruction with an RTS. An RTS executed with no previous JSR will cause execution to begin at whatever address happens to be pulled from the stack.

SC – Set Counter

**Mnemonic**: SC

**Opcode**:0101

**Usage**: SC Counter Value

The SC instruction is used to load a counter with a particular value. The operand determines which of counters C1 and C2 is to be loaded. The Value to be loaded is one of 2, 3, 4, 7, 10, 15, 19, and 31. The counter values are used for looping and indexing. Both C1 and C2 can be used for looping constructs (when combined with the DBR instruction), while only C1 can be used for indexing 32-bit parts of multi-precision variables.

SET – Set Bits

**Mnemonic**: SET

**Opcode**:0111

**Usage**: SET Flag/Register

The SET instruction allows the setting of particular flags or flash memory. There is also a corresponding CLR instruction. The WE and K2MX operands each set the specified flag for later processing. The IST and ISW operands each set the appropriate bit in Flash memory, while the MTR operand transfers the current value in the Accumulator into the MTR register. The SET Nx command loads N1 – N4 with the following constants:

| **Index** | **Constant Loaded** | **Initial X[N] referred to** |
|---|---|---|
| N1 | 2 | X[13] |
| N2 | 7 | X[8] |
| N3 | 13 | X[2] |
| N4 | 15 | X[0] |

Note that each initial $X[N_n]$ referred to matches the optimized SHA-1 algorithm initial states for indexes N1 – N4. When each index value $N_n$ decrements, the effective X[N] increments. This is because the X words are stored in memory with most significant word first.

ST – Store Accumulator

**Mnemonic**: ST

**Opcode**:1111

**Usage**: ST Location

The ST instruction is stores the current value of the Accumulator in the specified location. The location is one of A, B, C, D, E, T, AM, MT, AE[C1], H[C1], B160[C1], R[C1], K[C1], M[C1], or X[N4]. The X[N4] operand has the side effect of advancing the N4 index. After the store has taken place, N4 will be pointing to the next element in the X array. N4 decrements by 1, but since the X array is ordered from high to low, to decrement the index advances to the next element in the array. If the destination is in Flash memory, the effect of the ST instruction is to set the bits in the Flash memory corresponding to the bits in the Accumulator. To ensure a store of the exact value from the Accumulator, be sure to use the CLR instruction to erase the appropriate memory location first.

TBR – Test and Branch

**Mnemonic:** TBR

**Opcode:** 0000

**Usage:** TBR Value Index

The Test and Branch instruction tests whether the Accumulator is zero or non-zero, and then branches to the given address if the Accumulator's current state matches that being tested for. If the Z flag matches the TRB test, replace the PC by 9 bit value where bit0 = 0 and upper 8 bits come from MU. Otherwise increment current PC by 1. The Value operand is either 0 or 1. A 0 indicates the test is for the Accumulator to be zero. A 1 indicates the test is for the Accumulator to be non-zero. The Index operand indicates where execution is to jump to should the test succeed. The remaining 3 bits of operand index into the lowest 8 entries of Jump Table 1. The upper 8 bits are taken from the table, and the lowest bit (bit 0) is cleared to 0. CMD is cleared to 0 upon a RESET. 0 is translated as TBR 0, which means branch to the address stored in address offset 0 if the Accumulator = 0. Since the Accumulator and Z flag are also cleared to 0 on a RESET, the test will be true, so the net effect is a jump to the address stored in the 0th entry in the jump table.

XOR – Exclusive OR

**Mnemonic:** XOR

**Opcode:** 1010

**Usage:** XOR Value

The XOR instruction performs a 32-bit bitwise XOR with the Accumulator, and stores the result in the Accumulator. The operand is one of A, B, C, D, E, T, AM, MT, X[N1], X[N2], X[N3], or X[N4]. The Z flag is also set during this operation, depending on the result (i.e. what value is loaded into the Accumulator). A bitwise NOT operation can be performed by XORing the Accumulator with 0xFFFFFFFF (via the LDK instruction). The X[N] operands have a side effect of advancing the appropriate index to the next value (after the operation). After the XOR has taken place, the index will be pointing to the next element in the X array. N4 is also advanced by the ST X[N4] instruction. The index decrements by 1, but since the X array is ordered from high to low, to decrement the index advances to the next element in the array.

ProgrammingMode Detection Unit

The ProgrammingMode Detection Unit monitors the input clock voltage. If the clock voltage is a particular value the Erase Tamper Detection Line is triggered to erase all keys, program code, secret information etc and enter Program Mode. The ProgrammingMode Detection Unit can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS. There is no particular need to cover the ProgrammingMode Detection Unit by the Tamper Detection Lines, since an attacker can always place the chip in ProgrammingMode via the CLK input. The use of the Erase Tamper Detection Line as the signal for entering Programming Mode means that if an attacker wants to use Programming Mode as part of an attack, the Erase Tamper Detection Lines must be active and functional. This makes an attack on the Authentication Chip far more difficult.

Noise Generator

The Noise Generator can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS. However, the Noise Generator must be protected by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information. In addition, the bits in the LFSR must be validated to ensure they have not been tampered with (i.e. a parity check). If the parity check fails, the Erase Tamper Detection Line is triggered. Finally, all 64 bits of the Noise Generator are ORed into a single bit. If this bit is 0, the Erase Tamper Detection Line is triggered. This is because 0 is an invalid state for an LFSR. There is no point in using an OK bit setup since the Noise Generator bits are only used by the Tamper Detection and Prevention circuitry.

State Machine

The State Machine is responsible for generating the two operating cycles of the CPU, stalling during long command operations, and storing the op-code and operand during operating cycles. The State Machine can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS. However, the opcode/operand latch needs to be parity-checked. The logic and registers contained in the State Machine must be covered by both Tamper Detection Lines. This is to ensure that the instructions to be executed are not changed by an attacker.

The Authentication Chip does not require the high speeds and throughput of a general purpose CPU. It must operate fast enough to perform the authentication protocols, but not faster. Rather than have specialized circuitry for optimizing branch control or executing opcodes while fetching the next one (and all the complexity associated with that), the state machine adopts a simplistic view of the world. This helps to minimize design time as well as reducing the possibility of error in implementation.

The general operation of the state machine is to generate sets of cycles:

*Cycle 0:* ***Fetch cycle***. This is where the opcode is fetched from the program memory, and the effective address from the fetched opcode is generated.

*Cycle 1:* ***Execute cycle***. This is where the operand is (potentially) looked up via the generated effective address (from Cycle 0) and the operation itself is executed.

Under normal conditions, the state machine generates cycles: 0, 1, 0, 1, 0, 1, 0, 1... However, in some cases, the state machine stalls, generating Cycle 0 each clock tick until the stall condition finishes. Stall conditions include waiting for erase cycles of Flash memory, waiting for clients to read or write serial information, or an invalid opcode (due to tampering). If the Flash memory is currently being erased, the next instruction cannot execute until the Flash memory has finished being erased. This is determined by the ***Wait*** signal coming from the Memory Unit. If Wait = 1, the State Machine must only generate Cycle 0s. There are also two cases for stalling due to serial I/O operations:

The opcode is ROR OutBit,

from the MU, and OutBitValid and InBitValid come from the I/O Unit. When Cycle is 0, the 8-bit op-code is fetched from the memory unit and placed in the 8-bit CMD register. The write enable for the CMD register is therefore ~Cycle. There are two outputs from this unit: Cycle and CMD. Both of these values are passed into all the other processing units within the Authentication Chip. The 1-bit Cycle value lets each unit know whether a fetch or execute cycle is taking place, while the 8-bit CMD value allows each unit to take appropriate action for commands related to the specific unit.

Fig. 187 shows the data flow and relationship between components of the State Machine where:

| | |
|---|---|
| $Logic_1$: | Wait OR<br>~(Old OR ((CMD=ROR) & ((CMD=InBit AND ~InBitValid) OR<br>(CMD=OutBit AND OutBitValid)))) |

Old and CMD are both cleared to 0 upon a RESET. This results in the first cycle being 1, which causes the 0 CMD to be executed. 0 is translated as TBR 0, which means branch to the address stored in address offset 0 if the Accumulator = 0. Since the Accumulator is also cleared to 0 on a RESET, the test will be true, so the net effect is a jump to the address stored in the 0th entry in the jump table. The two VAL units are designed to validate the data that passes through them. Each contains an OK bit connected to both Tamper Prevention and Detection Lines. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. In the case of $VAL_1$, the effective Cycle will always be 0 if the chip has been tampered with. Thus no program code will execute since there will never be a Cycle 1. There is no need to check if Old has been tampered with, for if an attacker freezes the Old state, the chip will not execute any further instructions. In the case of $VAL_2$, the effective 8-bit CMD value will always be 0 if the chip has been tampered with, which is the TBR 0 instruction. This will stop execution of any program code. $VAL_2$ also performs a parity check on the bits from CMD to ensure that CMD has not been tampered with. If the parity check fails, the Erase Tamper Detection Line is triggered.

I/O Unit

The I/O Unit is responsible for communicating serially with the outside world. The Authentication Chip acts as a *slave* serial device, accepting serial data from a client, processing the command, and sending the resultant data to the client serially. The I/O Unit can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS. In addition, none of the latches need to be parity checked since there is no advantage for an attacker to destroy or modify them. The I/O Unit outputs 0s and inputs 0s if either of the Tamper Detection Lines is broken. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since breaking either Tamper Detection Lines should result in a RESET or the erasure of all Flash memory

The InBit, InBitValid, OutBit, and OutBitValid 1 bit registers are used for communication between the client (System) and the Authentication Chip. InBit and InBitValid provide the means for clients to pass commands and data to the

Authentication Chip. OutBit and OutBitValid provide the means for clients to get information from the Authentication Chip.When the chip is RESET, InBitValid and OutBitValid are both cleared. A client sends commands and parameter bits to the Authentication Chip one bit at a time. From the Authentication Chip's point of view:

Reads from InBit will hang while InBitValid is clear. InBitValid will remain clear until the client has written the next input bit to InBit. Reading InBit clears the InBitValid bit to allow the next InBit to be read from the client. A client cannot write a bit to the Authentication Chip unless the InBitValid bit is clear.

Writes to OutBit will hang while OutBitValid is set. OutBitValid will remain set until the client has read the bit from OutBit. Writing OutBit sets the OutBitValid bit to allow the next OutBit to be read by the client. A client cannot read a bit from the Authentication Chip unless the OutBitValid bit is set.

The actual stalling of commands is taken care of by the State Machine, but the various communication registers and the communication circuitry is found in the I/O Unit.

Fig. 188 shows the data flow and relationship between components of the I/O Unit where:

| | |
|---|---|
| $Logic_1$: | Cycle AND (CMD = ROR OutBit) |

The Serial I/O unit contains the circuitry for communicating externally with the external world via the Data pin. The InBitUsed control signal must be set by whichever unit consumes the InBit during a given clock cycle (which can be any state of Cycle). The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry, each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of $VAL_1$, the effective bit output from the chip will always be 0 if the chip has been tampered with. Thus no useful output can be generated by an attacker. In the case of $VAL_2$, the effective bit input to the chip will always be 0 if the chip has been tampered with. Thus no useful input can be chosen by an attacker. There is no need to verify the registers in the I/O Unit since an attacker does not gain anything by destroying or modifying them.

## ALU

Fig. 189 illustrates a schematic block diagram of the Arithmetic Logic Unit. The Arithmetic Logic Unit (ALU) contains a 32-bit Acc (**Accumulator**) register as well as the circuitry for simple arithmetic and logical operations. The ALU and all sub-units must be implemented with non-flashing CMOS since the key passes through it. In addition, the Accumulator must be parity-checked. The logic and registers contained in the ALU must be covered by both Tamper Detection Lines. This is to ensure that keys and intermediate calculation values cannot be changed by an attacker. A 1-bit Z register contains the state of zero-ness of the Accumulator. Both the Z and Accumulator registers are cleared to 0 upon a RESET. The Z register is updated whenever the Accumulator is updated, and the Accumulator is updated for any of the commands: LD, LDK, LOG, XOR, ROR, RPL, and ADD. Each arithmetic and logical block operates on two 32-bit inputs: the current value of the Accumulator, and the current 32-bit output of the MU. Where:

| | |
|---|---|
| $Logic_1$: | Cycle AND $CMD_7$ AND ($CMD_{6\text{-}4} \neq$ ST) |

Since the WriteEnables of Acc and Z takes $CMD_7$ and Cycle into account (due to $Logic_1$), these two bits are not

required by the multiplexor $MX_1$ in order to select the output. The output selection for $MX_1$ only requires bits 6-3 of CMD and is therefore simpler as a result.

| | **Output** | **$CMD_{6-3}$** |
|---|---|---|
| $MX_1$ | ADD | ADD |
| | AND | LOG AND |
| | OR | LOG OR |
| | XOR | XOR |
| | RPL | RPL |
| | ROR | ROR |
| | From MU | LD or LDK |

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry, each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. In the case of $VAL_1$, the effective bit output from the Accumulator will always be 0 if the chip has been tampered with. This prevents an attacker from processing anything involving the Accumulator. $VAL_1$ also performs a parity check on the Accumulator, setting the Erase Tamper Detection Line if the check fails. In the case of $VAL_2$, the effective Z status of the Accumulator will always be true if the chip has been tampered with. Thus no looping constructs can be created by an attacker. The remaining function blocks in the ALU are described as follows. All must be implemented in non-flashing CMOS.

| **Block** | **Description** |
|---|---|
| OR | Takes the 32-bit output from the multiplexor $MX_1$, ORs all 32 bits together to get 1 bit. |
| ADD | Outputs the result of the addition of its two inputs, modulo $2^{32}$. |
| AND | Outputs the 32-bit result of a parallel bitwise AND of its two 32-bit inputs. |
| OR | Outputs the 32-bit result of a parallel bitwise OR of its two 32-bit inputs. |
| XOR | Outputs the 32-bit result of a parallel bitwise XOR of its two 32-bit inputs. |
| RPL | Examined in further detail below. |
| ROR | Examined in further detail below. |

<u>RPL</u>

Fig. 190 illustrates a schematic block diagram of the RPL unit. The RPL unit is a component within the ALU. It is designed to implement the RPLCMP functionality of the Authentication Chip. The RPLCMP command is specifically designed for use in secure writing to Flash memory M, based upon the values in AccessMode. The RPL unit contains a 32-bit shift register called AMT (**A**ccess**M**ode**T**emp), which shifts right two bits each shift pulse, and two 1-bit registers called EE and DE, directly based upon the WR pseudocode's EqEncountered and DecEncountered flags. All

registers are cleared to 0 upon a RESET. AMT is loaded with the 32 bit AM value (via the Accumulator) with a RPL INIT command, and EE and DE are set according to the general write algorithm via calls to RPL MHI and RPL MLO. The EQ and LT blocks have functionality exactly as documented in the WR command pseudocode. The EQ block outputs 1 if the 2 16-bit inputs are bit-identical and 0 if they are not. The LT block outputs 1 if the upper 16-bit input from the Accumulator is less than the 16-bit value selected from the MU via $MX_2$. The comparison is unsigned. The bit patterns for the operands are specifically chosen to make the combinatorial logic simpler. The bit patterns for the operands are listed again here since we will make use of the patterns:

| Operand | $CMD_{3-0}$ |
|---|---|
| Init | 0000 |
| MLO | 1110 |
| MHI | 1111 |

The MHI and MLO have the hi bit set to easily differentiate them from the Init bit pattern, and the lowest bit can be used to differentiate between MHI and MLO. The EE and DE flags must be updated each time the RPL command is issued. For the Init stage, we need to setup the two values with 0, and for MHI and MLO, we need to update the values of EE and DE appropriately. The WriteEnable for EE and DE is therefore:

| | |
|---|---|
| $Logic_1$: | Cycle AND ($CMD_{7-4}$ = RPL) |

With the 32 bit AMT register, we want to load the register with the contents of AM (read from the MU) upon an RPL Init command, and to shift the AMT register right two bit positions for the RPL MLO and RPL MHI commands. This can be simply tested for with the highest bit of the RPL operand ($CMD_3$). The WriteEnable and ShiftEnable for the AMT register is therefore:

| | |
|---|---|
| $Logic_2$ | $Logic_1$ AND $CMD_3$ |
| $Logic_3$ | $Logic_1$ AND $\sim CMD_3$ |

The output from $Logic_3$ is also useful as input to multiplexor $MX_1$, since it can be used to gate through either the current 2 access mode bits or 00 (which results in a reset of the DE and EE registers since it represents the access mode RW). Consequently $MX_1$ is:

| | Output | $Logic_3$ |
|---|---|---|
| $MX_1$ | AMT output | 0 |
| | 00 | 1 |

The RPL logic only replaces the upper 16 bits of the Accumulator. The lower 16 bits pass through untouched.

However, of the 32 bits from the MU (corresponding to one of M[0-15]), only the upper or lower 16 bits are used. Thus $MX_2$ tests $CMD_0$ to distinguish between MHI and MLO.

| | **Output** | **$CMD_0$** |
|---|---|---|
| $MX_2$ | Lower 16 bits | 0 |
| | Upper 16 bits | 1 |

The logic for updating the DE and EE registers matches the pseudocode of the WR command. Note that an input of an AccessMode value of 00 (=RW which occurs during an RPL INIT) causes both DE and EE to be loaded with 0 (the correct initialization value). EE is loaded with the result from $Logic_4$, and DE is loaded with the result from$Logic_5$.

| | |
|---|---|
| $Logic_4$ | (((AccessMode=MSR) AND EQ) OR<br>((AccessMode=NMSR) AND EE AND EQ)) |
| $Logic_5$ | (((AccessMode=MSR) AND LT) OR<br>((AccessMode=NMSR) AND DE) OR<br>((AccessMode=NMSR) AND EQ AND LT)) |

The upper 16 bits of the Accumulator must be replaced with the value that is to be written to M. Consequently $Logic_6$ matches the WE flag from the WR command pseudocode.

| | |
|---|---|
| $Logic_6$ | ((AccessMode=RW) OR<br>((AccessMode=MSR) AND LT) OR<br>((AccessMode=NMSR) AND (DE OR LT))) |

The output from $Logic_6$ is used directly to drive the selection between the original 16 bits from the Accumulator and the value from M[0-15] via multiplexor $MX_3$. If the 16 bits from the Accumulator are selected (leaving the Accumulator unchanged), this signifies that the Accumulator value can be written to M[n]. If the 16-bit value from M is selected (changing the upper 16 bits of the Accumulator), this signifies that the 16-bit value in M will be unchanged. $MX_3$ therefore takes the following form:

| | **Output** | **$Logic_6$** |
|---|---|---|
| $MX_3$ | 16 bits from MU | 0 |
| | 16 bits from Acc | 1 |

There is no point parity checking AMT as an attacker is better off forcing the input to $MX_3$ to be 0 (thereby enabling an attacker to write any value to M). However, if an attacker is going to go to the trouble of laser-cutting the chip

(including all Tamper Detection tests and circuitry), there are better targets than allowing the possibility of a limited chosen-text attack by fixing the input of $MX_3$.

ROR

Fig. 191 illustrates a schematic block diagram of the ROR block of the ALU. The ROR unit is a component within the ALU. It is designed to implement the ROR functionality of the Authentication Chip. A 1-bit register named RTMP is contained within the ROR unit. RTMP is cleared to 0 on a RESET, and set during the ROR RB and ROR XRB commands. The RTMP register allows implementation of Linear Feedback Shift Registers with any tap configuration. The XOR block is a 2 single-bit input, 1-bit out XOR. The RORn, blocks are shown for clarity, but in fact would be hardwired into multiplexor $MX_3$, since each block is simply a rewiring of the 32-bits, rotated right N bits.All 3 multiplexors ($MX_1$, $MX_2$, and $MX_3$) depend upon the 8-bit CMD value. However, the bit patterns for the ROR op-code are arranged for logic optimization purposes. The bit patterns for the operands are listed again here since we will make use of the patterns:

| **Operand** | **$CMD_{3-0}$** |
|---|---|
| InBit | 0000 |
| OutBit | 0001 |
| RB | 0010 |
| XRB | 0011 |
| IST | 0100 |
| ISW | 0101 |
| MTRZ | 0110 |
| 1 | 0111 |
| 2 | 1001 |
| 27 | 1010 |
| 31 | 1100 |

$Logic_1$ is used to provide the WriteEnable signal to RTMP. The RTMP register should only be written to during ROR RB and ROR XRB commands. $Logic_2$ is used to provide the control signal whenever the InBit is consumed. The two combinatorial logic blocks are:

| | |
|---|---|
| $Logic_1$: | Cycle AND ($CMD_{7-4}$ = ROR) AND ($CMD_{3-1}$= 001) |
| $Logic_2$: | Cycle AND ($CMD_{7-0}$ = ROR InBit) |

With multiplexor $MX_1$, we are selecting the bit to be stored in RTMP. $Logic_1$ already narrows down the CMD inputs to one of RB and XRB. We can therefore simply test $CMD_0$ to differentiate between the two. The following table

expresses the relationship between $CMD_0$ and the value output from $MX_1$.

| | Output | $CMD_0$ |
|---|---|---|
| $MX_1$ | $Acc_0$ | 0 |
| | XOR output | 1 |

With multiplexor $MX_2$, we are selecting which input bit is going to replace bit 0 of the Accumulator input. We can only perform a small amount of optimization here, since each different input bit typically relates to a specific operand. The following table expresses the relationship between $CMD_{3-0}$ and the value output from $MX_2$.

| | Output | $CMD_{3-0}$ | Comment |
|---|---|---|---|
| $MX_2$ | $Acc_0$ | 1xxx OR 111 | 1, 2, 27, 31 |
| | RTMP | 001x | RB, XRB |
| | InBit | 000x | InBit, OutBit |
| | $MU_0$ | 010x | IST, ISW |
| | MTRZ | 110 | MTRZ |

The final multiplexor, $MX_3$, does the final rotating of the 32-bit value. Again, the bit patterns of the CMD operand are taken advantage of:

| | Output | $CMD_{3-0}$ | Comment |
|---|---|---|---|
| $MX_3$ | ROR 1 | 0xxx | All except 2, 27, and 31 |
| | ROR 2 | 1xx1 | 2 |
| | ROR 27 | 1x1x | 27 |
| | ROR 31 | 11xx | 31 |

MinTicks Unit

Fig. 192 shows the data flow and relationship between components of the MinTicks Unit. The MinTicks Unit is responsible for a programmable minimum delay (via a countdown) between key-based operations within the Authentication Chip. The logic and registers contained in the MinTicksUnit must be covered by both Tamper Detection Lines. This is to ensure that an attacker cannot change the time between calls to key-based functions. Nearly all of the MinTicks Unit can be implemented with regular CMOS, since the key does not pass through most of this unit. However the Accumulator is used in the SET MTR instruction. Consequently this tiny section of circuitry must be implemented in non-flashing CMOS. The remainder of the MinTicks Unit does not have to be implemented with non-flashing CMOS. However, the MTRZ latch (see below) needs to be parity checked.

The MinTicks Unit contains a 32-bit register named MTR (MinTicksRemaining). The MTR register contains the number of clock ticks remaining before the next key-based function can be called. Each cycle, the value in MTR is decremented by 1 until the value is 0. Once MTR hits 0, it does not decrement any further. An additional one-bit register named MTRZ (MinTicksRegisterZero) reflects the current zero-ness of the MTR register. MTRZ is 1 if the MTRZ register is 0, and MTRZ is 0 if the MTRZ register is not 0. The MTR register is cleared by a RESET, and set to a new count via the SET MTR command, which transfers the current value in the Accumulator into the MTR register. Where:

| | |
|---|---|
| $Logic_1$ | CMD = SET MTR |

And :

| | Output | $Logic_1$ | MTRZ |
|---|---|---|---|
| $MX_1$ | Acc | 1 | - |
| | MTR-1 | 0 | 0 |
| | 0 | 0 | 1 |

Since Cycle is connected to the WriteEnables of MTR and MTRZ, these registers only update during the Execute cycle, i.e. when Cycle = 1. The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry, each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. In the case of $VAL_1$, the effective output from MTR is 0, which means that the output from the decrementor unit is all 1s, thereby causing MTRZ to remain 0, thereby preventing an attacker from using the key-based functions. $VAL_1$ also validates the parity of the MTR register. If the parity check fails, the Erase Tamper Detection Line is triggered. In the case of $VAL_2$, if the chip has been tampered with, the effective output from MTRZ will be 0, indicating that the MinTicksRemaining register has not yet reached 0, thereby preventing an attacker from using the key-based functions.

**Program Counter Unit**

Fig. 192 is a block diagram of the Program Counter Unit. The Program Counter Unit (PCU) includes the 9 bit PC (Program Counter), as well as logic for branching and subroutine control. The Program Counter Unit can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS. However, the latches need to be parity-checked. In addition, the logic and registers contained in the Memory Unit must be covered by both Tamper Detection Lines to ensure that the PC cannot be changed by an attacker. The PC is actually implemented as a 6-level by 9-bit PCA (PC Array), indexed by the 3-bit SP (Stack Pointer) register. The PC and SP registers are all cleared to 0 on a RESET, and updated during the flow of program control according to the opcodes. The current value for the PC is output to the MU during Cycle 0 (the Fetch cycle). The PC is updated during Cycle 1 (the Execute cycle) according on the command being executed. In most cases, the PC simply increments by 1. However, when branching occurs (due to subroutine or some other form of jump), the PC is replaced by a new value. The mechanism for calculating the new PC value depends upon the opcode

being processed.

The ADD block is a simple adder modulo $2^9$. The inputs are the PC value and either 1 (for incrementing the PC by 1) or a 9 bit offset (with hi bit set and lower 8 bits from the MU). The "+1" block takes a 3-bit input and increments it by 1 (with wrap). The "-1" block takes a 3-bit input and decrements it by 1 (with wrap). The different forms of PC control are as follows:

| Command | Action |
|---|---|
| JSR,<br>JSI (ACC) | Save old value of PC onto stack for later.<br>New PC is 9 bit value where bit0 = 0 (subroutines must therefore start at an even address), and upper 8 bits of address come from MU<br>(MU 8-bit value is Jump Table 1 for JSR, and Jump Table 2 for JSI) |
| JSI RTS | Pop old value of PC from stack and increment by 1 to get new PC. |
| TBR | If the Z flag matches the TRB test, replace PC by 9 bit value where bit0 = 0 and upper 8 bits come from MU. Otherwise increment current PC by 1. |
| DBR C1,<br>DBR C2 | Add 9 bit offset (8 bit value from MU and hi bit = 1) to current PC only if the C1Z or C2Z is set (C1Z for DBR C1, C2Z for DBR C2). Otherwise increment current PC by 1. |
| All others | Increment current PC by 1. |

Since the same action takes place for JSR, and JSI (ACC), we specifically detect that case in $Logic_1$. By the same concept, we can specifically test for the JSI RTS case in $Logic_2$.

| | |
|---|---|
| $Logic_1$ | $(CMD_{7-5} = 001)$ OR $(CMD_{7-3} = 01001)$ |
| $Logic_2$ | $CMD_{7-3} = 01000$ |

When updating the PC, we must decide if the PC is to be replaced by a completely new item, or by the result of the adder. This is the case for JSR and JSI (ACC), as well as TBR as long as the test bit matches the state of the Accumulator. All but TBR is tested for by $Logic_1$, so $Logic_3$ also includes the output of $Logic_1$ as its input. The output from $Logic_3$ is then used by multiplexors $MX_2$ to obtain the new PC value.

| | |
|---|---|
| $Logic_3$ | $Logic_1$ OR<br>$((CMD_{7-4} = TBR)$ AND $(CMD_3$ XOR $Z))$ |

| | Output | $Logic_3$ |
|---|---|---|
| $MX_2$ | Output from Adder | 0 |
| | Replacement value | 1 |

The input to the 9-bit adder depends on whether we are incrementing by 1 (the usual case), or adding the offset as read from the MU (the DBR command). $Logic_4$ generates the test. The output from $Logic_4$ is then directly used by multiplexor $MX_3$ accordingly.

| | |
|---|---|
| $Logic_4$ | (($CMD_{7-3}$ = DBR C1) AND C1Z) OR<br>($CMD_{7-3}$ = DBR C2) AND C2Z)) |

| | Output | $Logic_4$ |
|---|---|---|
| $MX_3$ | Output from Adder | 0 |
| | Replacement value | 1 |

Finally, the selection of which PC entry to use depends on the current value for SP. As we enter a subroutine, the SP index value must increment, and as we return from a subroutine, the SP index value must decrement. In all other cases, and when we want to fetch a command (Cycle 0), the current value for the SP must be used. $Logic_1$ tells us when a subroutine is being entered, and $Logic_2$ tells us when the subroutine is being returned from. The multiplexor selection is therefore defined as follows:

| | Output | Cycle/$Logic_1$/$Logic_2$ |
|---|---|---|
| | SP-1 | 1x1 |
| $MX_1$ | SP+1 | 11x |
| | SP | 0xx OR 00 |

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry), each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. Both VAL units also parity-check the data bits to ensure that they are valid. If the parity-check fails, the Erase Tamper Detection Line is triggered. In the case of $VAL_1$, the effective output from the SP register will always be 0. If the chip has been tampered with. This prevents an attacker from executing any subroutines.In the case of $VAL_2$, the effective PC output will always be 0 if the chip has been tampered with. This prevents an attacker from executing any program code.

**Memory Unit**

The Memory Unit (MU) contains the internal memory of the Authentication Chip. The internal memory is addressed

by 9 bits of address, which is passed in from the Address Generator Unit. The Memory Unit outputs the appropriate 32-bit and 8-bit values according to the address. The Memory Unit is also responsible for the special Programming Mode, which allows input of the program Flash memory. The contents of the entire Memory Unit must be protected from tampering. Therefore the logic and registers contained in the Memory Unit must be covered by both Tamper Detection Lines. This is to ensure that program code, keys, and intermediate data values cannot be changed by an attacker. All Flash memory needs to be multi-state, and must be checked upon being read for invalid voltages. The 32-bit RAM also needs to be parity-checked. The 32-bit data paths through the Memory Unit must be implemented with non-flashing CMOS since the key passes along them. The 8-bit data paths can be implemented in regular CMOS since the key does not pass along them.

Constants

The Constants memory region has address range: 000000000 – 000001111. It is therefore the range 00000xxxx. However, given that the next 48 addresses are reserved, this can be taken advantage of during decoding. The Constants memory region can therefore be selected by the upper 3 bits of the address ($Adr_{8-6}$ = 000), with the lower 4 bits fed into combinatorial logic, with the 4 bits mapping to 32-bit output values as follows:

| $Adr_{3-0}$ | Output Value |
|---|---|
| 0000 | 0x00000000 |
| 0001 | 0x36363636 |
| 0010 | 0x5C5C5C5C |
| 0011 | 0xFFFFFFFF |
| 0100 | 0x5A827999 |
| 0101 | 0x6ED9EBA1 |
| 0110 | 0x8F1BBCDC |
| 0111 | 0xCA62C1D6 |
| 1000 | 0x67452301 |
| 1001 | 0xEFCDAB89 |
| 1010 | 0x98BADCFE |
| 1011 | 0x10325476 |
| 11xx | 0xC3D2E1F0 |

RAM

The address space for the 32 entry 32-bit RAM is 001000000 – 001011111. It is therefore the range 0010xxxxx. The RAM memory region can therefore be selected by the upper 4 bits of the address ($Adr_{8-5}$ = 0010), with the lower 5 bits selecting which of the 32 values to address. Given the contiguous 32-entry address space, the RAM can easily be implemented as a simple 32x32-bit RAM. Although the CPU treats each address from the range 00000 – 11111 in

special ways, the RAM address decoder itself treats no address specially. All RAM values are cleared to 0 upon a RESET, although any program code should not take this for granted.

Flash Memory – Variables

The address space for the 32-bit wide Flash memory is 001100000 – 001111111. It is therefore the range 0011xxxxx. The Flash memory region can therefore be selected by the upper 4 bits of the address ($Adr_{8-5}$ = 0111), with the lower 5 bits selecting which value to address. The Flash memory has special requirements for erasure. It takes quite some time for the erasure of Flash memory to complete. The Wait signal is therefore set inside the Flash controller upon receipt of a CLR command, and is only cleared once the requested memory has been erased. Internally, the erase lines of particular memory ranges are tied together, so that only 2 bits are required as indicated by the following table:

| $Adr_{4-3}$ | Erases range |
|---|---|
| 00 | $R_{0-4}$ |
| 01 | MT, AM, $K1_{0-4}$, $K2_{0-4}$ |
| 10 | Individual M address (Adr) |
| 11 | IST, ISW |

Flash values are unchanged by a RESET, although program code should not take the initial values for Flash (after manufacture) other than garbage. Operations that make use of Flash addresses are LD, ST, ADD, RPL, ROR, CLR, and SET. In all cases, the operands and the memory placement are closely linked, in order to minimize the address generation and decoding.The entire variable section of Flash memory is also erased upon entering Programming Mode, and upon detection of a definite physical Attack.

Flash Memory – Program

The address range for the 384 entry 8-bit wide program Flash memory is 010000000 – 111111111. It is therefore the range 01xxxxxxx – 11xxxxxxx. Decoding is straightforward given the ROM start address and address range. Although the CPU treats parts of the address range in special ways, the address decoder itself treats no address specially. Flash values are unchanged by a RESET, and are cleared only by entering Programming Mode. After manufacture, the Flash contents must be considered to be garbage. The 384 bytes can only be loaded by the State machine when in Programming Mode.

Block Diagram of MU

Fig. 193 is a block diagram of the Memory Unit. The logic shown takes advantage of the fact that 32-bit data and 8-bit data are required by separate commands, and therefore fewer bits are required for decoding. As shown, 32-bit output and 8-bit output are always generated. The appropriate components within the remainder of the Authentication Chip simply use the 32-bit or 8-bit value depending on the command being executed. Multiplexor $MX_1$, selects the 32-bit output from a choice of Truth Table constants, RAM, and Flash memory. Only 2 bits are required to select between these 3 outputs, namely $Adr_6$ and $Adr_5$. Thus $MX_2$takes the following form:

| | **Output** | **$Adr_{6\text{-}5}$** |
|---|---|---|
| $MX_2$ | Output from 32-bit Truth Table | 00 |
| | Output from 32-bit Flash memory | 10 |
| | Output from 32-bit RAM | 11 |

The logic for erasing a particular part of the 32-bit Flash memory is satisfied by $Logic_1$. The Erase Part control signal should only be set during a CLR command to the correct part of memory while Cycle=1. Note that a single CLR command may clear a range of Flash memory. $Adr_6$ is sufficient as an address range for CLR since the range will always be within Flash for valid operands, and 0 for non-valid operands. The *entire* range of 32-bit wide Flash memory is erased when the Erase Detection Lines is triggered (either by an attacker, or by deliberately entering Programming Mode).

| | |
|---|---|
| $Logic_1$ | Cycle AND ($CMD_{7\text{-}4}$ = CLR) AND $Adr_6$ |

The logic for writing to a particular part of Flash memory is satisfied by $Logic_2$. The WriteEnable control signal should only be set during an appropriate ST command to a Flash memory range while Cycle=1. Testing only $Adr_{6\text{-}5}$ is acceptable since the ST command only validly writes to Flash or RAM (if $Adr_{6\text{-}5}$ is 00, K2MX must be 0).

| | |
|---|---|
| $Logic_2$ | Cycle AND ($CMD_{7\text{-}4}$ = ST) AND ($Adr_{6\text{-}5}$ = 10) |

The WE (WriteEnable) flag is set during execution of the SET WE and CLR WE commands. $Logic_3$ tests for these two cases. The actual bit written to WE is $CMD_4$.

| | |
|---|---|
| $Logic_3$ | Cycle AND ($CMD_{7\text{-}5}$ = 011) AND ($CMD_{3\text{-}0}$ = 0000) |

The logic for writing to the RAM region of memory is satisfied by $Logic_4$. The WriteEnable control signal should only be set during an appropriate ST command to a RAM memory range while Cycle=1. However this is tempered by the WE flag, which governs whether writes to X[N] are permitted. The X[N] range is the upper half of the RAM, so this can be tested for using $Adr_4$. Testing only $Adr_{6\text{-}5}$ as the full address range of RAM is acceptable since the ST command only writes to Flash or RAM.

| | |
|---|---|
| $Logic_4$ | Cycle AND ($CMD_{7\text{-}4}$ = ST) AND ($Adr_{6\text{-}5}$ = 11) AND<br>(($Adr_4$ AND WE) OR ($\sim Adr_4$)) |

The three VAL units are validation units connected to the Tamper Prevention and Detection circuitry, each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each

cycle. The OK bit is ANDed with each data bit that passes through the unit. The VAL units also check the data bits to ensure that they are valid. $VAL_1$ and $VAL_2$ validate by checking the state of each data bit, and $VAL_3$ performs a parity check. If any validity test fails, the Erase Tamper Detection Line is triggered. In the case of $VAL_1$, the effective output from the program Flash will always be 0 (interpreted as TBR 0) if the chip has been tampered with. This prevents an attacker from executing any useful instructions. In the case of $VAL_2$, the effective 32-bit output will always be 0 if the chip has been tampered with. Thus no key or intermediate storage value is available to an attacker. The 8-bit Flash memory is used to hold the program code, jump tables and other program information. The 384 bytes of Program Flash memory are selected by the full 9 bits of address (using address range 01xxxxxxx – 11xxxxxxx). The Program Flash memory is erased only when the Erase Detection Lines is triggered (either by an attacker, or by entering Programming Mode due to the Programming Mode Detection Unit). When the Erase Detection Line is triggered, a small state machine in the Program Flash Memory Unit erases the 8-bit Flash memory, validates the erasure, and loads in the new contents (384 bytes) from the serial input. The following pseudocode illustrates the state machine logic that is executed when the Erase Detection line is triggered:

```
Set WAIT output bit to prevent the remainder of the chip from functioning
Fix 8-bit output to be 0
Erase all 8-bit Flash memory
Temp ← 0
For Adr = 0 to 383
    Temp ← Temp OR Flash_Adr
IF (Temp ≠ 0)
    Hang
For Adr = 0 to 383
    Do 8 times
                        Wait for InBitValid to be set
                        ShiftRight[Temp, InBit]
    Set InBitUsed control signal
    Flash_Adr ← Temp
Hang
```

During the Programming Mode state machine execution, 0 must be placed onto the 8-bit output. A 0 command causes the remainder of the Authentication chip to interpret the command as a TBR 0. When the chip has read all 384 bytes into the Program Flash Memory, it hangs (loops indefinitely). The Authentication Chip can then be reset and the program used normally. Note that the erasure is validated by the same 8-bit register that is used to load the new contents of the 8-bit program Flash memory. This helps to reduce the chances of a successful attack, since program code can't be loaded properly if the register used to validate the erasure is destroyed by an attacker. In addition, the entire state machine is protected by both Tamper Detection lines.

**Address Generator Unit**

The Address Generator Unit generates effective addresses for accessing the Memory Unit (MU). In Cycle 0, the PC is passed through to the MU in order to fetch the next opcode. The Address Generator interprets the returned opcode in order to generate the effective address for Cycle 1. In Cycle 1, the generated address is passed to the MU. The logic and registers contained in the Address Generator Unit must be covered by both Tamper Detection Lines. This is to ensure that an attacker cannot alter any generated address. Nearly all of the Address Generator Unit can be implemented with regular CMOS, since the key does not pass through most of this unit. However 5 bits of the Accumulator are used in the JSI Address generation. Consequently this tiny section of circuitry must be implemented in non-flashing CMOS. The remainder of the Address Generator Unit does not have to be implemented with non-flashing CMOS. However, the latches for the counters and calculated address should be parity-checked. If either of the Tamper Detection Lines is broken, the Address Generator Unit will generate address 0 each cycle and all counters will be fixed at 0. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since under normal circumstances, breaking a Tamper Detection Line will result in a RESET or the erasure of all Flash memory.

Background to Address Generation

The logic for address generation requires an examination of the various opcodes and operand combinations. The relationship between opcode/operand and address is examined in this section, and is used as the basis for the Address Generator Unit.

Constants

The lower 4 entries are the simple constants for general-purpose use as well as the HMAC algorithm. The lower 4 bits of the LDK operand directly correspond to the lower 3 bits of the address in memory for these 4 values, i.e. 0000, 0001, 0010, and 0011 respectively. The y constants and the h constants are also addressed by the LDK command. However the address is generated by ORing the lower 3 bits of the operand with the inverse of the C1 counter value, and keeping the 4th bit of the operand intact. Thus for LDK y, the y operand is 0100, and with LDK h, the h operand is 1000. Since the inverted C1 value takes on the range 000 – 011 for y, and 000 – 100 for h, the ORed result gives the exact address. For all constants, the upper 5 bits of the final address are always 00000.

RAM

Variables A-T have addresses directly related to the lower 3 bits of their operand values. That is, for operand values 0000 – 0101 of the LD, ST, ADD, LOG, and XOR commands, as well as operand vales 1000-1101 of the LOG command, the lower 3 operand address bits can be used together with a constant high 6-bit address of 001000 to generate the final address. The remaining register values can only be accessed via an indexed mechanism. Variables A-E, B160, and H are only accessible as indexed by the C1 counter value, while X is indexed by $N_1$, $N_2$, $N_3$, and $N_4$. With the LD, ST and ADD commands, the address for AE as indexed by C1 can be generated by taking the lower 3 bits of the operand (000) and ORing them with the C1 counter value. However, H and B160 addresses cannot be generated in this way, (otherwise the RAM address space would be non-contiguous). Therefore simple combinatorial logic must convert AE into 0000, H into 0110, and B160 into 1011. The final address can be obtained by adding C1 to the 4-bit value (yielding a 4-bit result), and prepending the constant high 5-bit address of 00100. Finally, the X range

of registers is only accessed as indexed by $N_1$, $N_2$, $N_3$, and $N_4$. With the XOR command, any of $N_{1-4}$ can be used to index, while with LD, ST, and ADD, only $N_4$ can be used. Since the operand of X in LD, ST, and ADD is the same as the $X_{N4}$ operand, the lower 2 bits of the operand selects which N to use. The address can thus be generated as a constant high 5-bit value of 00101, with the lower 4 bits coming from by the selected N counter.

Flash Memory – Variables

The addresses for variables MT and AM can be generated from the operands of associated commands. The 4 bits of operand can be used directly (0110 and 0111), and prepending the constant high 5-bit address of 00110. Variables $R_{1-5}$, $K1_{1-5}$, $K2_{1-5}$, and $M_{0-7}$ are only accessible as indexed by the inverse of the C1 counter value (and additionally in the case of R, by the actual C1 value). Simple combinatorial logic must convert R and RF into 00000, K into 01000 or 11000 depending on whether K1 or K2 is being addressed, and M (including MHI and MLO) into 10000. The final address can be obtained by ORing (or adding) C1 (or in the case of RF, using C1 directly) with the 5-bit value, and prepending the constant high 4-bit address of 0011. Variables IST and ISW are each only 1 bit of value, but can be implemented by any number of bits. Data is read and written as either 0x00000000 or 0xFFFFFFFF. They are addressed only by ROR, CLR and SET commands. In the case of ROR, the low bit of the operand is combined with a constant upper 8-bits value of 00111111, yielding 001111110 and 001111111 for IST and ISW respectively. This is because none of the other ROR operands make use of memory, so in cases other than IST and ISW, the value returned can be ignored. With SET and CLR, IST and ISW are addressed by combining a constant upper 4-bits of 0011 with a mapping from IST (0100) to 11110 and from ISW (0101) to 11111. Since IST and ISW share the same operand values with E and T from RAM, the same decoding logic can be used for the lower 5 bits. The final address requires bits 4, 3, and 1 to be set (this can be done by ORing in the result of testing for operand values 010x).

Flash Memory – Program

The address to lookup in program Flash memory comes directly from the 9-bit PC (in Cycle 0) or the 9-bit Adr register (in Cycle 1). Commands such as TBR, DBR, JSR and JSI modify the PC according to data stored in tables at specific addresses in the program memory. As a result, address generation makes use of some constant address components, with the command operand (or the Accumulator) forming the lower bits of the effective address:

| **Command** | **Address Range** | **Constant (upper) part of address** | **Variable (lower) part of address** |
|---|---|---|---|
| TBR | 010000xxx | 010000 | $CMD_{2-0}$ |
| JSR | 0100xxxxx | 0100 | $CMD_{4-0}$ |
| JSI ACC | 0101xxxxx | 0101 | $Acc_{4-0}$ |
| DBR | 011000xxx | 011000 | $CMD_{2-0}$ |

Block Diagram of Address Generator Unit

Fig. 194 shows a schematic block diagram for the Address Generator Unit. The primary output from the Address Generator Unit is selected by multiplexor $MX_1$, as shown in the following table:

| | **Output** | **Cycle** |
|---|---|---|
| $MX_1$ | PC | 0 |
| | Adr | 1 |

It is important to distinguish between the CMD data and the 8-bit data from the MU:

In Cycle 0, the 8-bit data line holds the next instruction to be executed in the following Cycle 1. This 8-bit command value is used to decode the effective address. By contrast, the CMD 8-bit data holds the previous instruction, so should be ignored.

In Cycle 1, the CMD line holds the currently executing instruction (which was in the 8-bit data line during Cycle 0), while the 8-bit data line holds the data at the effective address from the instruction. The CMD data must be executed during Cycle 1.

Consequently, the choice of 9-bit data from the MU or the CMD value is made by multiplexor MX3, as shown in the following table:

| | **Output** | **Cycle** |
|---|---|---|
| $MX_3$ | 8-bit data from MU | 0 |
| | CMD | 1 |

Since the 9-bit Adr register is updated every Cycle 0, the WriteEnable of Adr is connected to ~Cycle. The Counter Unit generates counters C1, C2 (used internally) and the selected N index. In addition, the Counter Unit outputs flags C1Z and C2Z for use by the Program Counter Unit. The various *GEN units generate addresses for particular command types during Cycle 0, and multiplexor $MX_2$ selects between them based on the command as read from program memory via the PC (i.e. the 8-bit data line). The generated values are as follows:

| **Block** | **Commands for which address is generated** |
|---|---|
| JSIGEN | JSI ACC |
| JSRGEN | JSR, TBR |
| DBRGEN | DBR |
| LDKGEN | LDK |
| RPLGEN | RPL |
| VARGEN | LD, ST, ADD, LOG, XOR |
| BITGEN | ROR, SET |
| CLRGEN | CLR |

Multiplexor $MX_2$ has the following selection criteria:

| | Output | 8-bit data value from MU |
|---|---|---|
| $MX_2$ | 9-bit value from JSIGEN | 01001xxx |
| | 9-bit value from JSRGEN | 001xxxxx OR 0000xxxx |
| | 9-bit value from DBRGEN | 0001xxxx |
| | 9-bit value from LDKGEN | 1110xxxx |
| | 9 bit value from RPLGEN | 1101xxxx |
| | 9-bit value from VARGEN | 10xxxxxx OR 1x11xxxx |
| | 9-bit value from BITGEN | 0111xxxx OR 1100xxxx |
| | 9 bit value from CLRGEN | 0110xxxx |

The $VAL_1$ unit is a validation unit connected to the Tamper Prevention and Detection circuitry. It contains an OK bit that is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with the 9 bits of Effective Address before they can be used. If the chip has been tampered with, the address output will be always 0, thereby preventing an attacker from accessing other parts of memory. The $VAL_1$ unit also performs a parity check on the Effective Address bits to ensure it has not been tampered with. If the parity-check fails, the Erase Tamper Detection Line is triggered.

JSIGEN

Fig. 195 shows a schematic block diagram for the JSIGEN Unit. The JSIGEN Unit generates addresses for the JSI ACC instruction. The effective address is simply the concatenation of:

the 4-bit high part of the address for the JSI Table (0101) and

the lower 5 bits of the Accumulator value.

Since the Accumulator may hold the key at other times (when a jump address is not being generated), the value must be hidden from sight. Consequently this unit must be implemented with non-flashing CMOS. The multiplexor $MX_1$ simply chooses between the lower 5 bits from Accumulator or 0, based upon whether the command is JSIGEN. Multiplexor $MX_1$ has the following selection criteria:

| | Output | $CMD_{7-0}$ |
|---|---|---|
| $MX_1$ | $Accumulator_{4-0}$ | JSI ACC |
| | 00000 | ~(JSI ACC) |

JSRGEN

Fig. 196 shows a schematic block diagram for the JSRGEN Unit. The JSRGEN Unit generates addresses for the JSR and TBR instructions. The effective address comes from the concatenation of:

the 4-bit high part of the address for the JSR table (0100),

the offset within the table from the operand (5 bits for JSR commands, and 3 bits plus a constant 0 bit for TBR).

where $Logic_1$ produces bit 3 of the effective address. This bit should be bit 3 in the case of JSR, and 0 in the case of TBR:

| | |
|---|---|
| $Logic_1$ | $bit_5$ AND $bit_3$ |

Since the JSR instruction has a 1 in bit 5, (while TBR is 0 for this bit) ANDing this with bit 3 will produce bit 3 in the case of JSR, and 0 in the case of TBR.

DBRGEN

Fig. 197 shows a schematic block diagram for the DBRGEN Unit. The DBRGEN Unit generates addresses for the DBR instructions. The effective address comes from the concatenation of:

the 6-bit high part of the address for the DBR table (011000), and

the lower 3 bits of the operand

LDKGEN

Fig. 198 shows a schematic block diagram for the LDKGEN Unit. The LDKGEN Unit generates addresses for the LDK instructions. The effective address comes from the concatenation of:

the 5-bit high part of the address for the LDK table (00000),

the high bit of the operand, and

the lower 3 bits of the operand (in the case of the lower constants), or the lower 3 bits of the operand ORed with C1 (in the case of indexed constants).

The $OR_2$ block simply ORs the 3 bits of C1 with the 3 lowest bits from the 8-bit data output from the MU. The multiplexor $MX_1$ simply chooses between the actual data bits and the data bits ORed with C1, based upon whether the upper bits of the operand are set or not. The selector input to the multiplexor is a simple OR gate, ORing $bit_2$ with $bit_3$. Multiplexor $MX_1$ has the following selection criteria:

| | **Output** | **$bit_3$ OR $bit_2$** |
|---|---|---|
| $MX_1$ | $bit_{2-0}$ | 0 |
| | Output from OR block | 1 |

RPLGEN

Fig. 199 shows a schematic block diagram for the RPLGEN Unit. The RPLGEN Unit generates addresses for the RPL instructions. When K2MX is 0, the effective address is a constant 000000000. When K2MX is 1 (indicating reads from M return valid values), the effective address comes from the concatenation of:

the 6-bit high part of the address for M (001110), and

the 3 bits of the current value for C1

The multiplexor $MX_1$ chooses between the two addresses, depending on the current value of K2MX. Multiplexor $MX_1$ therefore has the following selection criteria:

| | Output | K2MX |
|---|---|---|
| $MX_1$ | 000000000 | 0 |
| | 001110 \| C1 | 1 |

VARGEN

Fig. 200 shows a schematic block diagram for the VARGEN Unit. The VARGEN Unit generates addresses for the LD, ST, ADD, LOG, and XOR instructions. The K2MX 1-bit flag is used to determine whether reads from M are mapped to the constant 0 address (which returns 0 and cannot be written to), and which of K1 and K2 is accessed when the operand specifies K. The 4-bit Adder block takes 2 sets of 4-bit inputs, and produces a 4-bit output via addition modulo $2^4$. The single bit register K2MX is only ever written to during execution of a CLR K2MX or a SET K2MX instruction. $Logic_1$ sets the K2MX WriteEnable based on these conditions:

| | |
|---|---|
| $Logic_1$ | Cycle AND $bit_{7-0}$=011x0001 |

The bit written to the K2MX variable is 1 during a SET instruction, and 0 during a CLR instruction. It is convenient to use the low order bit of the opcode ($bit_4$) as the source for the input bit. During address generation, a Truth Table implemented as combinatorial logic determines part of the base address as follows:

| $bit_{7-4}$ | $bit_{3-0}$ | Description | Output Value |
|---|---|---|---|
| LOG | x | A, B, C, D, E, T, MT, AM | 00000 |
| ≠ LOG | 0xxx OR 1x00 | A, B, C, D, E, T, MT, AM, AE[C1], R[C1] | 00000 |
| ≠ LOG | 1001 | B160 | 01011 |
| ≠ LOG | 1010 | H | 00110 |
| ≠ LOG | 111x | X, M | 10000 |
| ≠ LOG | 1101 | K | K2MX \| 1000 |

Although the Truth Table produces 5 bits of output, the lower 4 bits are passed to the 4-bit Adder, where they are added to the index value (C1, N or the lower 3 bits of the operand itself). The highest bit passes the adder, and is prepended to the 4-bit result from the adder result in order to produce a 5-bit result. The second input to the adder comes from multiplexor $MX_1$, which chooses the index value from C1, N, and the lower 3 bits of the operand itself). Although C1 is only 3 bits, the fourth bit is a constant 0. Multiplexor $MX_1$ has the following selection criteria:

| | Output | $bit_{7-0}$ |
|---|---|---|
| $MX_1$ | $Data_{2-0}$ | $(bit_3=0)$ OR $(bit_{7-4}=LOG)$ |
| | C1 | $(bit_3=1)$ AND $(bit_{2-0}\neq 111)$ AND $((bit_{7-4}=1x11)$ OR $(bit_{7-4}=ADD))$ |
| | N | $((bit_3=1)$ AND $(bit_{7-4}=XOR))$ OR $(((bit_{7-4}=1x11)$ OR $(bit_{7-4}=ADD))$ AND $(bit_{3-0}=1111))$ |

The 6th bit ($bit_5$) of the effective address is 0 for RAM addresses, and 1 for Flash memory addresses. The Flash memory addresses are MT, AM, R, K, and M. The computation for $bit_5$ is provided by $Logic_2$:

| | |
|---|---|
| $Logic_2$ | $((bit_{3-0}=110)$ OR $(bit_{3-0}=011x)$ OR $(bit_{3-0}=110x))$ AND $((bit_{7-4}=1x11)$ OR $(bit_{7-4}=ADD))$ |

A constant 1 bit is prepended, making a total of 7 bits of effective address. These bits will form the effective address unless K2MX is 0 and the instruction is LD, ADD or ST M[C1]. In the latter case, the effective address is the constant address of 0000000. In both cases, two 0 bits are prepended to form the final 9-bit address. The computation is shown here, provided by $Logic_3$ and multiplexor $MX_2$.

| | |
|---|---|
| $Logic_3$ | ~K2MX AND $(bit_{3-0}=1110)$ AND $((bit_{7-4}=1x11)$ OR $(bit_{7-4}=ADD))$ |

| | Output | $Logic_3$ |
|---|---|---|
| $MX_2$ | Calculated bits | 0 |
| | 0000000 | 1 |

## CLRGEN

Fig. 201 shows a schematic block diagram for the CLRGEN Unit. The CLRGEN Unit generates addresses for the CLR instruction. The effective address is always in Flash memory for valid memory accessing operands, and is 0 for invalid operands. The CLR M[C1] instruction *always* erases M[C1], regardless of the status of the K2MX flag (kept in the VARGEN Unit). The Truth Table is simple combinatorial logic that implements the following relationship:

| **Input Value ($bit_{3-0}$)** | **Output Value** |
|---|---|
| 1100 | 00 1100 000 |
| 1101 | 00 1101 000 |
| 1110 | 00 1110 \| C1 |
| 1111 | 00 1111 110 |
| ~(11xx) | 000000000 |

It is a simple matter to reduce the logic required for the Truth Table since in all 4 main cases, the first 6 bits of the effective address are 00 followed by the operand ($bits_{3-0}$).

BITGEN

Fig. 202 shows a schematic block diagram for the BITGEN Unit. The BITGEN Unit generates addresses for the ROR and SET instructions. The effective address is always in Flash memory for valid memory accessing operands, and is 0 for invalid operands. Since ROR and SET instructions only access the IST and ISW Flash memory addresses (the remainder of the operands access registers), a simple combinatorial logic Truth Table suffices for address generation:

| **Input Value ($bit_{3-0}$)** | **Output Value** |
|---|---|
| 010x | 00111111 \| $bit_0$ |
| ~(010x) | 000000000 |

Counter Unit

Fig. Y37 shows a schematic block diagram for the Counter Unit. The Counter Unit generates counters C1, C2 (used internally) and the selected N index. In addition, the Counter Unit outputs flags C1Z and C2Z for use externally. Registers C1 and C2 are updated when they are the targets of a DBR or SC instruction. The high bit of the operand ($bit_3$ of the effective command) gives the selection between C1 and C2. $Logic_1$ and $Logic_2$ determine the WriteEnables for C1 and C2 respectively.

| | |
|---|---|
| $Logic_1$ | Cycle AND ($bit_{7-3}$=0x010) |
| $Logic_2$ | Cycle AND ($bit_{7-3}$=0x011) |

The single bit flags C1Z and C2Z are produced by the NOR of their multibit C1 and C2 counterparts. Thus C1Z is 1 if C1 = 0, and C2Z is 1 if C2 = 0. During a DBR instruction, the value of either C1 or C2 is decremented by 1 (with wrap). The input to the Decrementor unit is selected by multiplexor $MX_2$ as follows:

| | Output | $bit_3$ |
|---|---|---|
| $MX_2$ | C1 | 0 |
| | C2 | 1 |

The actual value written to C1 or C2 depends on whether the DBR or SC instruction is being executed. Multiplexor $MX_1$ selects between the output from the Decrementor (for a DBR instruction), and the output from the Truth Table (for a SC instruction). Note that only the lowest 3 bits of the 5-bit output are written to C1. Multiplexor $MX_1$ therefore has the following selection criteria:

| | Output | $bit_6$ |
|---|---|---|
| $MX_1$ | Output from Truth Table | 0 |
| | Output from Decrementor | 1 |

The Truth Table holds the values to be loaded by C1 and C2 via the SC instruction. The Truth Table is simple combinatorial logic that implements the following relationship:

| Input Value ($bit_{2-0}$) | Output Value |
|---|---|
| 000 | 00010 |
| 001 | 00011 |
| 010 | 00100 |
| 011 | 00111 |
| 100 | 01010 |
| 101 | 01111 |
| 110 | 10011 |
| 111 | 11111 |

Registers N1, N2, N3, and N4 are updated by their next value – 1 (with wrap) when they are referred to by the XOR instruction. Register N4 is also updated when a ST X[N4] instruction is executed. LD and ADD instructions do not update N4. In addition, all 4 registers are updated during a SET Nx command. $Logic_{4-7}$ generate the WriteEnables for registers N1-N4. All use $Logic_3$, which produces a 1 if the command is SET Nx, or 0 otherwise.

| | |
|---|---|
| $Logic_3$ | $bit_{7-0}$=01110010 |
| $Logic_4$ | Cycle AND (($bit_{7-0}$=10101000) OR $Logic_3$) |
| $Logic_5$ | Cycle AND (($bit_{7-0}$=10101001) OR $Logic_3$) |
| $Logic_6$ | Cycle AND (($bit_{7-0}$=10101010) OR $Logic_3$) |
| $Logic_7$ | Cycle AND (($bit_{7-0}$=11111011) OR ($bit_{7-0}$=10101011) OR $Logic_3$) |

The actual N index value passed out, or used as the input to the Decrementor, is simply selected by multiplexor $MX_4$ using the lower 2 bits of the operand:

| | **Output** | **$bit_{1-0}$** |
|---|---|---|
| $MX_4$ | N1 | 00 |
| | N2 | 01 |
| | N3 | 10 |
| | N4 | 11 |

The Incrementor takes 4 bits of input value (selected by multiplexor $MX_4$) and adds 1, producing a 4-bit result (due to addition modulo $2^4$). Finally, four instances of multiplexor $MX_3$ select between a constant value (different for each N, and to be loaded during the SET Nx command), and the result of the Decrementor (during XOR or ST instructions). The value will only be written if the appropriate WriteEnable flag is set (see $Logic_4$ - $Logic_7$), so $Logic_3$ can safely be used for the multiplexor.

| | **Output** | **$Logic_3$** |
|---|---|---|
| $MX_3$ | Output from Decrementor | 0 |
| | Constant value | 1 |

The SET Nx command loads N1 – N4 with the following constants:

| **Index** | **Constant Loaded** | **Initial X[N] referred to** |
|---|---|---|
| N1 | 2 | X[13] |
| N2 | 7 | X[8] |
| N3 | 13 | X[2] |
| N4 | 15 | X[0] |

Note that each initial $X[N_n]$ referred to matches the optimized SHA-1 algorithm initial states for indexes N1 – N4.

When each index value $N_n$ decrements, the effective X[N] increments. This is because the X words are stored in memory with most significant word first. The three VAL units are validation units connected to the Tamper Prevention and Detection circuitry, each with an OK bit. The OK bit is set to 1 on RESET, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. All VAL units also parity check the data to ensure the counters have not been tampered with. If a parity check fails, the Erase Tamper Detection Line is triggered. In the case of $VAL_1$, the effective output from the counter C1 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs that index through the keys. In the case of $VAL_2$, the effective output from the counter C2 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs. In the case of $VAL_3$, the effective output from any N counter (N1-N4) will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs that index through X.

Turning now to Fig. 203, there is illustrated 705 the information stored within the flash memory store 701. This data can include the following:

Factory Code

The factory code is a 16 bit code indicating the factory at which the print roll was manufactured. This identifies factories belonging to the owner of the print roll technology, or factories making print rolls under license. The purpose of this number is to allow the tracking of factory that a print roll came from, in case there are quality problems.

Batch Number

The batch number is a 32 bit number indicating the manufacturing batch of the print roll. The purpose of this number is to track the batch that a print roll came from, in case there are quality problems.

Serial Number

A 48 bit serial number is provided to allow unique identification of each print roll up to a maximum of 280 trillion print rolls.

Manufacturing date

A 16 bit manufacturing date is included for tracking the age of print rolls, in case the shelf life is limited.

Media length

The length of print media remaining on the roll is represented by this number. This length is represented in small units such as millimeters or the smallest dot pitch of printer devices using the print roll and to allow the calculation of the number of remaining photos in each of the well known C, H, and P formats, as well as other formats which may be printed. The use of small units also ensures a high resolution can be used to maintain synchronization with pre-printed media.

Media Type

The media type datum enumerates the media contained in the print roll.

(1) Transparent

(2) Opaque white

(3) Opaque tinted

(4) 3D lenticular

(5) Pre-printed: length specific

(6) Pre-printed: not length specific

(7) Metallic foil

(8) Holographic/optically variable device foil

Pre-printed Media Length

The length of the repeat pattern of any pre-printed media contained, for example on the back surface of the print roll is stored here.

Ink Viscosity

The viscosity of each ink color is included as an 8 bit number. the ink viscosity numbers can be used to adjust the print head actuator characteristics to compensate for viscosity (typically, a higher viscosity will require a longer actuator pulse to achieve the same drop volume).

Recommended Drop Volume for 1200 dpi

The recommended drop volume of each ink color is included as an 8 bit number. The most appropriate drop volume will be dependent upon the ink and print media characteristics. For example, the required drop volume will decrease with increasing dye concentration or absorptivity. Also, transparent media require around twice the drop volume as opaque white media, as light only passes through the dye layer once for transparent media.

As the print roll contains both ink and media, a custom match can be obtained. The drop volume is only the recommended drop volume, as the printer may be other than 1200 dpi, or the printer may be adjusted for lighter or darker printing.

Ink Color

The color of each of the dye colors is included and can be used to "fine tune" the digital half toning that is applied to any image before printing.

Remaining Media Length Indicator

The length of print media remaining on the roll is represented by this number and is updatable by the camera device. The length is represented in small units (eg. 1200 dpi pixels) to allow calculation of the number of remaining photos in each of C, H, and P formats, as well as other formats which may be printed. The high resolution can also be used to maintain synchronization with pre-printed media.

Copyright or Bit Pattern

This 512 bit pattern represents an ASCII character sequence sufficient to allow the contents of the flash memory store to be copyrightable.

Turning now to Fig. 204, there is illustrated the storage table 730 of the Artcam authorization chip. The table includes manufacturing code, batch number and serial number and date which have an identical format to that previously described. The table 730 also includes information 731 on the print engine within the Artcam device. The information stored can include a print engine type, the DPI resolution of the printer and a printer count of the number of prints produced by the printer device.

Further, an authentication test key 710 is provided which can randomly vary from chip to chip and is utilised as the Artcam random identification code in the previously described algorithm. The 128 bit print roll authentication key 713 is also provided and is equivalent to the key stored within the print rolls. Next, the 512 bit pattern is stored

followed by a 120 bit spare area suitable for Artcam use.

As noted previously, the Artcam preferably includes a liquid crystal display 15 which indicates the number of prints left on the print roll stored within the Artcam. Further, the Artcam also includes a three state switch 17 which allows a user to switch between three standard formats C H and P (classic, HDTV and panoramic). Upon switching between the three states, the liquid crystal display 15 is updated to reflect the number of images left on the print roll if the particular format selected is used.

In order to correctly operate the liquid crystal display, the Artcam processor, upon the insertion of a print roll and the passing of the authentication test reads the from the flash memory store of the print roll chip 53 and determines the amount of paper left. Next, the value of the output format selection switch 17 is determined by the Artcam processor. Dividing the print length by the corresponding length of the selected output format the Artcam processor determines the number of possible prints and updates the liquid crystal display 15 with the number of prints left. Upon a user changing the output format selection switch 17 the Artcam processor 31 re-calculates the number of output pictures in accordance with that format and again updates the LCD display 15.

The storage of process information in the printer roll table 705 (Fig. 165) also allows the Artcam device to take advantage of changes in process and print characteristics of the print roll.

In particular, the pulse characteristics applied to each nozzle within the print head can be altered to take into account of changes in the process characteristics. Turning now to Fig. 205, the Artcam Processor can be adapted to run a software program stored in an ancillary memory ROM chip. The software program, a pulse profile characteriser 771 is able to read a number of variables from the printer roll. These variables include the remaining roll media on printer roll 772, the printer media type 773, the ink color viscosity 774, the ink color drop volume 775 and the ink color 776. Each of these variables are read by the pulse profile characteriser and a corresponding, most suitable pulse profile is determined in accordance with prior trial and experiment. The parameters alters the printer pulse received by each printer nozzle so as to improve the stability of ink output.

It will be evident that the authorization chip includes significant advances in that important and valuable information is stored on the printer chip with the print roll. This information can include process characteristics of the print roll in question in addition to information on the type of print roll and the amount of paper left in the print roll. Additionally, the print roll interface chip can provide valuable authentication information and can be constructed in a tamper proof manner. Further, a tamper resistant method of utilising the chip has been provided. The utilization of the print roll chip also allows a convenient and effective user interface to be provided for an immediate output form of Artcam device able to output multiple photographic formats whilst simultaneously able to provide an indicator of the number of photographs left in the printing device.

Print Head Unit

Turning now to Fig. 206 , there is illustrated an exploded perspective view, partly in section, of the print head unit 615 of Fig. 162.

The print head unit 615 is based around the print-head 44 which ejects ink drops on demand on to print media 611 so as to form an image. The print media 611 is pinched between two set of rollers comprising a first set 618, 616 and second set 617, 619.

The print-head 44 operates under the control of power, ground and signal lines 810 which provides power and control for the print-head 44 and are bonded by means of Tape Automated Bonding (TAB) to the surface of the print-

head 44.

Importantly, the print-head 44 which can be constructed from a silicon wafer device suitably separated, relies upon a series of anisotropic etches 812 through the wafer having near vertical side walls. The through wafer etches 812 allow for the direct supply of ink to the print-head surface from the back of the wafer for subsequent ejection.

The ink is supplied to the back of the inkjet print-head 44 by means of ink-head supply unit 814. The inkjet print-head 44 has three separate rows along its surface for the supply of separate colors of ink. The ink-head supply unit 814 also includes a lid 815 for the sealing of ink channels.

In Fig. 207 to Fig. 210, there is illustrated various perspective views of the ink-head supply unit 814. Each of Fig. 207 to Fig. 210 illustrate only a portion of the ink head supply unit which can be constructed of indefinite length, the portions shown so as to provide exemplary details. In Fig. 207 there is illustrated a bottom perspective view, Fig. 148 illustrates a top perspective view, Fig. 209 illustrates a close up bottom perspective view, partly in section, Fig. 210 illustrates a top side perspective view showing details of the ink channels, and Fig. 211 illustrates a top side perspective view as does Fig. 212.

There is considerable cost advantage in forming ink-head supply unit 814 from injection molded plastic instead of, say, micromachined silicon. The manufacturing cost of a plastic ink channel will be considerably less in volume and manufacturing is substantially easier. The design illustrated in the accompanying Figures assumes a 1600 dpi three color monolithic print head, of a predetermined length. The provided flow rate calculations are for a 100mm photo printer.

The ink-head supply unit 814 contains all of the required fine details. The lid 815 (Fig. 206) is permanently glued or ultrasonically welded to the ink-head supply unit 814 and provides a seal for the ink channels.

Turning to Fig. 209, the cyan, magenta and yellow ink flows in through ink inlets 820-822, the magenta ink flows through the throughholes 824,825 and along the magenta main channels 826,827 (Fig. 141). The cyan ink flows along cyan main channel 830 and the yellow ink flows along the yellow main channel 831. As best seen from Fig. 209, the cyan ink in the cyan main channels then flows into a cyan sub-channel 833. The yellow subchannel 834 similarly receiving yellow ink from the yellow main channel 831.

As best seen in Fig. 210, the magenta ink also flows from magenta main channels 826,827 through magenta throughholes 836, 837. Returning again to Fig. 209, the magenta ink flows out of the throughholes 836, 837. The magenta ink flows along first magenta subchannel e.g. 838 and then along second magenta subchannel e.g. 839 before flowing into a magenta trough 840. The magenta ink then flows through magenta vias e.g. 842 which are aligned with corresponding inkjet head throughholes (e.g. 812 of Fig. 166) wherein they subsequently supply ink to inkjet nozzles for printing out.

Similarly, the cyan ink within the cyan subchannel 833 flows into a cyan pit area 849 which supplies ink two cyan vias 843, 844. Similarly, the yellow subchannel 834 supplies yellow pit area 46 which in turn supplies yellow vias 847, 848.

As seen in Fig. 210, the print-head is designed to be received within print-head slot 850 with the various vias e.g. 851 aligned with corresponding through holes eg. 851 in the print-head wafer.

Returning to Fig. 206, care must be taken to provide adequate ink flow to the entire print-head chip 44, while satisfying the constraints of an injection moulding process. The size of the ink through wafer holes 812 at the back of the print head chip is approximately 100μm x 50μm, and the spacing between through holes carrying different colors

of ink is approximately 170μm. While features of this size can readily be molded in plastic (compact discs have micron sized features), ideally the wall height must not exceed a few times the wall thickness so as to maintain adequate stiffness. The preferred embodiment overcomes these problems by using hierarchy of progressively smaller ink channels.

In Fig. 211, there is illustrated a small portion 870 of the surface of the print-head 44. The surface is divided into 3 series of nozzles comprising the cyan series 871, the magenta series 872 and the yellow series 873. Each series of nozzles is further divided into two rows eg. 875, 876 with the print-head 44 having a series of bond pads 878 for bonding of power and control signals.

The print head is preferably constructed in accordance with a large number of different forms of ink jet invented for uses including Artcam devices. These ink jet devices are discussed in further detail hereinafter.

The print-head nozzles include the ink supply channels 880, equivalent to anisotropic etch hole 812 of Fig. 206. The ink flows from the back of the wafer through supply channel 881 and in turn through the filter grill 882 to ink nozzle chambers eg. 883. The operation of the nozzle chamber 883 and print-head 44 (Fig. 1) is, as mentioned previously, described in the abovementioned patent specification.

<u>Ink Channel Fluid Flow Analysis</u>

Turning now to an analysis of the ink flow, the main ink channels 826, 827, 830, 831 (Fig. 207, Fig. 141) are around 1mm x 1mm, and supply all of the nozzles of one color. The sub-channels 833, 834, 838, 839 (Fig. 209) are around 200μm x 100μm and supply about 25 inkjet nozzles each. The print head through holes 843, 844, 847, 848 and wafer through holes eg. 881 (Fig. 211) are 100μm x 50μm and, supply 3 nozzles at each side of the print head through holes. Each nozzle filter 882 has 8 slits, each with an area of 20μm x 2μm and supplies a single nozzle.

An analysis has been conducted of the pressure requirements of an ink jet printer constructed as described. The analysis is for a 1,600 dpi three color process print head for photograph printing. The print width was 100 mm which gives 6,250 nozzles for each color, giving a total of 18,750 nozzles.

The maximum ink flow rate required in various channels for full black printing is important. It determines the pressure drop along the ink channels, and therefore whether the print head will stay filled by the surface tension forces alone, or, if not, the ink pressure that is required to keep the print head full.

To calculate the pressure drop, a drop volume of 2.5 pl for 1,600 dpi operation was utilized. While the nozzles may be capable of operating at a higher rate, the chosen drop repetition rate is 5 kHz which is suitable to print a 150 mm long photograph in an little under 2 seconds. Thus, the print head, in the extreme case, has a 18,750 nozzles, all printing a maximum of 5,000 drops per second. This ink flow is distributed over the hierarchy of ink channels. Each ink channel effectively supplies a fixed number of nozzles when all nozzles are printing.

The pressure drop Δρ was calculated according to the Darcy-Weisbach formula:

$$\Delta\rho = \frac{\rho U^2 f L}{2D}$$

Where ρ is the density of the ink, $U$ is the average flow velocity, $L$ is the length, $D$ is the hydraulic diameter, and $f$ is a dimensionless friction factor calculated as follows:

$$f = \frac{k}{\mathrm{Re}}$$

Where Re is the Reynolds number and $k$ is a dimensionless friction coefficient dependent upon the cross section of the channel calculated as follows:

$$\mathrm{Re} = \frac{UD}{\nu}$$

Where ν is the kinematic viscosity of the ink.

For a rectangular cross section, $k$ can be approximated by:

$$k = \frac{64}{\frac{2}{3} + \frac{11b}{24a}\ \frac{11b}{24a}(2 - b/a)}$$

Where $a$ is the longest side of the rectangular cross section, and $b$ is the shortest side. The hydraulic diameter $D$ for a rectangular cross section is given by:

$$D = \frac{2ab}{a + b}$$

Ink is drawn off the main ink channels at 250 points along the length of the channels. The ink velocity falls linearly from the start of the channel to zero at the end of the channel, so the average flow velocity $U$ is half of the maximum flow velocity. Therefore, the pressure drop along the main ink channels is half of that calculated using the maximum flow velocity

Utilizing these formulas, the pressure drops can be calculated in accordance with the following tables:

Table of Ink Channel Dimensions and Pressure Drops

| | # of Items | Length | Width | Depth | Nozzles supplied | Max.ink flow at 5KHz(U) | Pressure drop Δρ |
|---|---|---|---|---|---|---|---|
| Central Moulding | 1 | 106mm | 6.4mm | 1.4mm | 18,750 | 0,23ml/s | NA |
| Cyan main channel (830) | 1 | 100mm | 1mm | 1mm | 6,250 | 0.16μl/μs | 111 Pa |
| Magenta main channel (826) | 2 | 100mm | 700μm | 700μm | 3,125 | 0.16μl/μs | 231 Pa |
| Yellow main channel (831) | 1 | 100mm | 1mm | 1mm | 6,250 | 0.16μl/μs | 111 Pa |
| Cyan sub-channel (833) | 250 | 1.5mm | 200μm | 100μm | 25 | 0.16μl/μs | 41.7 Pa |
| Magenta sub-channel (834)(a) | 500 | 200μm | 50μm | 100μm | 12.5 | 0,031μl/μs | 44.5 Pa |
| Magenta sub-channel (838)(b) | 500 | 400μm | 100μm | 200μm | 12.5 | 0.031μl/μs | 5.6 Pa |
| Yellow sub-channel (834) | 250 | 1.5mm | 200μm | 100μm | 25 | 0.016μl/μs | 41.7 Pa |
| Cyan pit (842) | 250 | 200μm | 100μm | 300μm | 25 | 0.010μl/μs | 3.2 Pa |
| Magenta through (840) | 500 | 200μm | 50μm | 200μm | 12.5 | 0.016μl/μs | 18.0 Pa |
| Yellow pit (846) | 250 | 200μm | 100μm | 300μm | 25 | 0.010μl/μs | 3.2 Pa |
| Cyan via (843) | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Magenta via (842) | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Yellow via | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Magenta through hole (837) | 500 | 200μm | 500μm | 100μm | 12.5 | 0.003μl/μs | 0.87 Pa |
| Chip slot | 1 | 100mm | 730μm | 625 | 18,750 | NA | NA |
| Print head through holes (881)(in the chip substrate) | 1500 | 600μ | 100μm | 50μm | 12.5 | 0.052μl/μs | 133 Pa |
| Print head channel segments | 1,000/ color | 50μm | 60μm | 20μm | 3.125 | 0.049μl/μs | 62.8 Pa |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (on chip front) | | | | | | | |
| Filter Slits (on entrance to nozzle chamber (882) | 8 per nozzle | 2μm | 2μm | 20μm | 0.125 | 0.039μl/μs | 251 Pa |
| Nozzle chamber (on chip front)(883) | 1 per nozzle | 70μm | 30μm | 20μm | 1 | 0.021μl/μs | 75.4 Pa |

The total pressure drop from the ink inlet to the nozzle is therefore approximately 701Pa for cyan and yellow, and 845 Pa for magenta. This is less than 1% of atmospheric pressure. Of course, when the image printed is less than full black, the ink flow (and therefore the pressure drop) is reduced from these values.

Making the Mould for the Ink-head Supply Unit

The ink head supply unit 14 (Fig. 1) has features as small as 50μ and a length of 106mm. It is impractical to machine the injection moulding tools in the conventional manner. However, even though the overall shape may be complex, there are no complex curves required. The injection moulding tools can be made using conventional milling for the main ink channels and other millimeter scale features, with a lithographically fabricated inset for the fine features. A LIGA process can be used for the inset.

A single injection moulding tool could readily have 50 or more cavities. Most of the tool complexity is in the inset.

Turning to Fig. 206, the printing system is constructed via moulding ink supply unit 814 and lid 815 together and sealing them together as previously described. Subsequently print-head 44 is placed in its corresponding slot 850. Adhesive sealing strips 852, 853 are placed over the magenta main channels so to ensure they are properly sealed. The Tape Automated Bonding (TAB) strip 810 is then connected to the inkjet print-head 44 with the tab bonding wires running in the cavity 855. As can best be seen from Fig. 206, Fig. 207 and Fig. 212, aperture slots 855 - 862 are provided for the snap in insertion of rollers. The slots provided for the "clipping in" of the rollers with a small degree of play subsequently being provided for simple rotation of the rollers.

In Fig. 213 to Fig. 217, there are illustrated various perspective views of the internal portions of a finally assembled Artcam device with devices appropriately numbered.

- Fig. 213 illustrates a top side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;
- Fig. 214 illustrates a bottom side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;Fig. 215 illustrates a first
- top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Fig. 216 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

- Fig. 217 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Postcard Print Rolls

Turning now to Fig. 218, in one form of the preferred embodiment, the output printer paper 11 can, on the side that is not to receive the printed image, contain a number of pre-printed "postcard" formatted backing portions 885. The postcard formatted sections 885 can include prepaid postage "stamps" 886 which can comprise a printed authorization from the relevant postage authority within whose jurisdiction the print roll is to be sold or utilised. By agreement with the relevant jurisdictional postal authority, the print rolls can be made available having different postages. This is especially convenient where overseas travelers are in a local jurisdiction and wishing to send a number of postcards to their home country. Further, an address format portion 887 is provided for the writing of address dispatch details in the usual form of a postcard. Finally, a message area 887 is provided for the writing of a personalized information.

Turning now to Fig. 218 and Fig. 219, the operation of the camera device is such that when a series of images 890-892 is printed on a first surface of the print roll, the corresponding backing surface is that illustrated in Fig. 218. Hence, as each image eg. 891 is printed by the camera, the back of the image has a ready made postcard 885 which can be immediately dispatched at the nearest post office box within the jurisdiction. In this way, personalized postcards can be created.

It would be evident that when utilising the postcard system as illustrated in Fig. 219 and Fig. 220 only predetermined image sizes are possible as the synchronization between the backing postcard portion 885 and the front image 891 must be maintained. This can be achieved by utilising the memory portions of the authentication chip stored within the print roll to store details of the length of each postcard backing format sheet 885. This can be achieved by either having each postcard the same size or by storing each size within the print rolls on-board print chip memory.

The Artcam camera control system can ensure that, when utilising a print roll having pre-formatted postcards, that the printer roll is utilised only to print images such that each image will be on a postcard boundary. Of course, a degree of "play" can be provided by providing border regions at the edges of each photograph which can account for slight misalignment.

Turning now to Fig. 220, it will be evident that postcard rolls can be pre-purchased by a camera user when traveling within a particular jurisdiction where they are available. The postcard roll can, on its external surface, have printed information including country of purchase, the amount of postage on each postcard, the format of each postcard (for example being C,H or P or a combination of these image modes), the countries that it is suitable for use with and the postage expiry date after which the postage is no longer guaranteed to be sufficient can also be provided.

Hence, a user of the camera device can produce a postcard for dispatch in the mail by utilising their hand held camera to point at a relevant scene and taking a picture having the image on one surface and the pre-paid postcard details on the other. Subsequently, the postcard can be addressed and a short message written on the postcard before its immediate dispatch in the mail.

In respect of the software operation of the Artcam device, although many different software designs are possible, in one design, each Artcam device can consist of a set of loosely coupled functional modules utilised in a coordinated way by a single embedded application to serve the core purpose of the device. While the functional modules are reused in different combinations in various classes of Artcam device, the application is specific to the class of Artcam device.

Most functional modules contain both software and hardware components. The software is shielded from details of the hardware by a hardware abstraction layer, while users of a module are shielded from its software implementation by an abstract software interface. Because the system as a whole is driven by user-initiated andhardware- initiated events, most modules can run one or more asynchronous event-driven processes.

The most important modules which comprise the generic Artcam device are shown in Fig. 221. In this and subsequent diagrams, software components are shown on the left separated by a vertical dashed line 901 from hardware components on the right. The software aspects of these modules are described below:

Software Modules - Artcam Application 902

The Artcam Application implements the high-level functionality of the Artcam device. This normally involves capturing an image, applying an artistic effect to the image, and then printing the image. In a camera-oriented Artcam device, the image is captured via the Camera Manager 903. In a printer-oriented Artcam device, the image is captured via the Network Manager 904, perhaps as the result of the image being "squirted" by another device.

Artistic effects are found within the unified file system managed by the File Manager 905. An artistic effect consist of a script file and a set of resources. The script is interpreted and applied to the image via the Image Processing Manager 906. Scripts are normally shipped on ArtCards known as Artcards. By default the application uses the script contained on the currently mounted Artcard.

The image is printed via the Printer Manager 908.

When the Artcam device starts up, the bootstrap process starts the various manager processes before starting the application. This allows the application to immediately request services from the various managers when it starts.

On initialization the application 902 registers itself as the handler for the events listed below. When it receives an event, it performs the action described in the table.

| User interface event | Action |
|---|---|
| Lock Focus | Perform any automatic pre-capture setup via the Camera Manager. This includes auto-focussing, auto-adjusting exposure, and charging the flash. This is normally initiated by the user pressing the Take button halfway. |
| Take | Capture an image via the Camera Manager. |
| Self-Timer | Capture an image in self-timed mode via the Camera Manager. |
| Flash Mode | Update the Camera Manager to use the next flash mode. Update the Status Display to show the new flash mode. |
| Print | Print the current image via the Printer Manager. Apply an artistic effect to the image via the Image Processing Manager if there is a current script. Update the remaining prints count on the Status Display (see *Print Roll Inserted* below). |
| Hold | Apply an artistic effect to the current image via the Image Processing Manager if there is a current script, but don't print the image. |
| Eject ArtCards | Eject the currently inserted ArtCards via the File Manager. |
| Print Roll Inserted | Calculate the number of prints remaining based on the Print Manager's remaining media length and the Camera Manager's aspect ratio. Update the remaining prints count on the Status display. |
| Print Roll Removed | Update the Status Display to indicate there is no print roll present. |

Where the camera includes a display, the application also constructs a graphical user interface via the User Interface Manager 910 which allows the user to edit the current date and time, and other editable camera parameters. The application saves all persistent parameters in flash memory.

Real-Time Microkernel 911

The Real-Time Microkernel schedules processes preemptively on the basis of interrupts and process priority. It provides integrated inter-process communication and timer services, as these are closely tied to process scheduling. All other operating system functions are implemented outside the microkernel.

Camera Manager 903

The Camera Manager provides image capture services. It controls the camera hardware embedded in the Artcam. It provides an abstract camera control interface which allows camera parameters to be queried and set, and images captured. This abstract interface decouples the application from details of camera implementation. The Camera Manager utilizes the following input / output parameters and commands:

| output parameters | domains |
|---|---|
| focus range | real, real |
| zoom range | real, real |
| aperture range | real, real |
| shutter speed range | real, real |

| input parameters | domains |
|---|---|
| focus | real |
| zoom | real |
| aperture | real |
| shutter speed | real |
| aspect ratio | *classic, HDTV, panoramic* |
| focus control mode | multi-point auto, single-point auto, manual |
| exposure control mode | *auto, aperture priority, shutter priority, manual* |
| flash mode | *auto, auto with red-eye removal, fill, off* |
| view scene mode | *on, off* |

| commands | return value domains |
|---|---|
| Lock Focus | none |
| Self-Timed Capture | Raw Image |
| Capture Image | Raw Image |

The Camera Manager runs as an asynchronous event-driven process. It contains a set of linked state machines, one for each asynchronous operation. These include auto focussing, charging the flash, counting down the self-timer, and capturing the image. On initialization the Camera Manager sets the camera hardware to a known state. This includes setting a normal focal distance and retracting the zoom. The software structure of the Camera Manager is illustrated in Fig. 222. The software components are described in the following subsections:

Lock Focus 913

Lock Focus automatically adjusts focus and exposure for the current scene, and enables the flash if necessary, depending on the focus control mode, exposure control mode and flash mode. Lock Focus is normally initiated in response to the user pressing the Take button halfway. It is part of the normal image capture sequence, but may be separated in time from the actual capture of the image, if the user *holds* the take button halfway depressed. This allows the user to do spot focusing and spot metering.

Capture Image 914

Capture Image captures an image of the current scene. It lights a red-eye lamp if the flash mode includes red-eye removal, controls the shutter, triggers the flash if enabled, and senses the image through the image sensor. It

determines the orientation of the camera, and hence the captured image, so that the image can be properly oriented during later image processing. It also determines the presence of camera motion during image capture, to trigger deblurring during later image processing.

Self-Timed Capture 915

Self-Timed Capture captures an image of the current scene after counting down a 20s timer. It gives the user feedback during the countdown via the self-timer LED. During the first 15s it can light the LED. During the last 5s it flashes the LED.

View Scene 917

View Scene periodically senses the current scene through the image sensor and displays it on the color LCD, giving the user an LCD-based viewfinder.

Auto Focus 918

Auto Focus changes the focal length until selected regions of the image are sufficiently sharp to signify that they are in focus. It assumes the regions are in focus if an image sharpness metric derived from specified regions of the image sensor is above a fixed threshold. It finds the optimal focal length by performing a gradient descent on the derivative of sharpness by focal length, changing direction and stepsize as required. If the focus control mode is *multi-point auto*, then three regions are used, arranged horizontally across the field of view. If the focus control mode is *single-point auto*, then one region is used, in the center of the field of view. Auto Focus works within the available focal length range as indicated by the focus controller. In fixed-focus devices it is therefore effectively disabled.

Auto Flash 919

Auto Flash determines if scene lighting is dim enough to require the flash. It assumes the lighting is dim enough if the scene lighting is below a fixed threshold. The scene lighting is obtained from the lighting sensor, which derives a lighting metric from a central region of the image sensor. If the flash is required, then it charges the flash.

Auto Exposure 920

The combination of scene lighting, aperture, and shutter speed determine the exposure of the captured image. The desired exposure is a fixed value. If the exposure control mode is *auto*, Auto Exposure determines a combined aperture and shutter speed which yields the desired exposure for the given scene lighting. If the exposure control mode is *aperture priority*, Auto Exposure determines a shutter speed which yields the desired exposure for the given scene lighting and current aperture. If the exposure control mode is *shutter priority*, Auto Exposure determines an aperture which yields the desired exposure for the given scene lighting and current shutter speed. The scene lighting is obtained from the lighting sensor, which derives a lighting metric from a central region of the image sensor.

Auto Exposure works within the available aperture range and shutter speed range as indicated by the aperture controller and shutter speed controller. The shutter speed controller and shutter controller hide the absence of a mechanical shutter in most Artcam devices.

If the flash is enabled, either manually or by Auto Flash, then the *effective* shutter speed is the duration of the flash, which is typically in the range 1/1000 s to 1/10000 s.

Image Processing Manager 906 (Fig. 221)

The Image Processing Manager provides image processing and artistic effects services. It utilises the VLIW Vector Processor embedded in the Artcam to perform high-speed image processing. The Image Processing Manager contains an interpreter for scripts written in the Vark image processing language. An artistic effect therefore consists of

a Vark script file and related resources such as fonts, clip images etc. The software structure of the Image Processing Manager is illustrated in more detail in Fig. 223 and include the following modules:

Convert and Enhance Image 921

The Image Processing Manager performs image processing in the device-independent CIE LAB color space, at a resolution which suits the reproduction capabilities of the Artcam printer hardware. The captured image is first enhanced by filtering out noise. It is optionally processed to remove motion-induced blur. The image is then converted from its device-dependent RGB color space to the CIE LAB color space. It is also rotated to undo the effect of any camera rotation at the time of image capture, and scaled to the working image resolution. The image is further enhanced by scaling its dynamic range to the available dynamic range.

Detect Faces 923

Faces are detected in the captured image based on hue and local feature analysis. The list of detected face regions is used by the Vark script for applying face-specific effects such as warping and positioning speech balloons.

Vark Image Processing Language Interpreter 924

Vark consists of a general-purpose programming language with a rich set of image processing extensions. It provides a range of primitive data types (integer, real, boolean, character), a range of aggregate data types for constructing more complex types (array, string, record), a rich set of arithmetic and relational operators, conditional and iterative control flow (if-then-else, while-do), and recursive functions and procedures. It also provides a range of image-processing data types (image, clip image, matte, color, color lookup table, palette, dither matrix, convolution kernel, etc.), graphics data types (font, text, path), a set of image-processing functions (color transformations, compositing, filtering, spatial transformations and warping, illumination, text setting and rendering), and a set of higher-level artistic functions (tiling, painting and stroking).

A Vark program is portable in two senses. Because it is interpreted, it is independent of the CPU and image processing engines of its host. Because it uses a device-independent model space and a device-independent color space, it is independent of the input color characteristics and resolution of the host input device, and the output color characteristics and resolution of the host output device.

The Vark Interpreter 924 parses the source statements which make up the Vark script and produces a parse tree which represents the semantics of the script. Nodes in the parse tree correspond to statements, expressions, sub-expressions, variables and constants in the program. The root node corresponds to the main procedure statement list.
The interpreter executes the program by executing the root statement in the parse tree. Each node of the parse tree asks its children to evaluate or execute themselves appropriately. An *if* statement node, for example, has three children - a condition expression node, a *then* statement node, and an *else* statement node. The if statement asks the condition expression node to evaluate itself, and depending on the boolean value returned asks the then statement or the else statement to execute itself. It knows nothing about the actual condition expression or the actual statements.

While operations on most data types are executed during execution of the parse tree, operations on *image* data types are deferred until after execution of the parse tree. This allows imaging operations to be optimized so that only those intermediate pixels which contribute to the final image are computed. It also allows the final image to be computed in multiple passes by spatial subdivision, to reduce the amount of memory required.

During execution of the parse tree, each imaging function simply returns an imaging graph - a graph whose nodes are imaging operators and whose leaves are images - constructed with its corresponding imaging operator as the

root and its image parameters as the root's children. The image parameters are of course themselves image graphs. Thus each successive imaging function returns a deeper imaging graph.

After execution of the parse tree, an imaging graph is obtained which corresponds to the final image. This imaging graph is then executed in a depth-first manner (like any expression tree), with the following two optimizations: (1) only those pixels which contribute to the final image are computed at a given node, and (2) the children of a node are executed in the order which minimizes the amount of memory required. The imaging operators in the imaging graph are executed in the optimized order to produce the final image. Compute-intensive imaging operators are accelerated using the VLIW Processor embedded in the Artcam device. If the amount of memory required to execute the imaging graph exceeds available memory, then the final image region is subdivided until the required memory no longer exceeds available memory.

For a well-constructed Vark program the first optimization is unlikely to provide much benefit *per se*. However, if the final image region is subdivided, then the optimization is likely to provide considerable benefit. It is precisely this optimization, then, that allows subdivision to be used as an effective technique for reducing memory requirements. One of the consequences of deferred execution of imaging operations is that program control flow cannot depend on image content, since image content is not known during parse tree execution. In practice this is not a severe restriction, but nonetheless must be borne in mind during language design.

The notion of deferred execution (or *lazy evaluation*) of imaging operations is described by Guibas and Stolfi (Guibas, L.J., and J. Stolfi, "A Language for Bitmap Manipulation", *ACM Transactions on Graphics*, Vol. 1, No. 3, July 1982, pp. 191-214). They likewise construct an imaging graph during the execution of a program, and during subsequent graph evaluation propagate the *result* region *backwards* to avoid computing pixels which do not contribute to the final image. Shantzis additionally propagates regions of *available* pixels *forwards* during imaging graph evaluation (Shantzis, M.A., "A Model for Efficient and Flexible Image Computing", *Computer Graphics Proceedings, Annual Conference Series*, 1994, pp. 147-154). The Vark Interpreter uses the more sophisticated multi-pass bi-directional region propagation scheme described by Cameron (Cameron, S., "Efficient Bounds in Constructive Solid Geometry", *IEEE Computer Graphics & Applications*, Vol. 11, No. 3, May 1991, pp. 68-74). The optimization of execution order to minimise memory usage is due to Shantzis, but is based on standard compiler theory (Aho, A.V., R. Sethi, and J.D. Ullman, "Generating Code from DAGs", in *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, pp. 557-567,). The Vark Interpreter uses a more sophisticated scheme than Shantzis, however, to support variable-sized image buffers. The subdivision of the result region in conjunction with region propagation to reduce memory usage is also due to Shantzis.

Printer Manager 908 (Fig. 221)

The Printer Manager provides image printing services. It controls the Ink Jet printer hardware embedded in the Artcam. It provides an abstract printer control interface which allows printer parameters to be queried and set, and images printed. This abstract interface decouples the application from details of printer implementation and includes the following variables:

| output parameters | domains |
|---|---|
| media is present | bool |
| media has fixed page size | bool |
| media width | real |
| remaining media length | real |
| fixed page size | real, real |

| input parameters | domains |
|---|---|
| page size | real, real |

| commands | return value domains |
|---|---|
| Print Image | none |

| output events |
|---|
| invalid media |
| media exhausted |
| media inserted |
| media removed |

The Printer Manager runs as an asynchronous event-driven process. It contains a set of linked state machines, one for each asynchronous operation. These include printing the image and auto mounting the print roll. The software structure of the Printer Manager is illustrated in Fig. 224. The software components are described in the following description:

Print Image 930

Print Image prints the supplied image. It uses the VLIW Processor to prepare the image for printing. This includes converting the image color space to device-specific CMY and producing half-toned bi-level data in the format expected by the print head.

Between prints, the paper is retracted to the lip of the print roll to allow print roll removal, and the nozzles can be capped to prevent ink leakage and drying. Before actual printing starts, therefore, the nozzles are uncapped and cleared, and the paper is advanced to the print head. Printing itself consists of transferring line data from the VLIW processor, printing the line data, and advancing the paper, until the image is completely printed. After printing is complete, the paper is cut with the guillotine and retracted to the print roll, and the nozzles are capped. The remaining media length is then updated in the print roll.

Auto Mount Print Roll 131

Auto Mount Print Roll responds to the insertion and removal of the print roll. It generates print roll insertion and removal events which are handled by the application and used to update the status display. The print roll is

authenticated according to a protocol between the authentication chip embedded in the print roll and the authentication chip embedded in Artcam. If the print roll fails authentication then it is rejected. Various information is extracted from the print roll. Paper and ink characteristics are used during the printing process. The remaining media length and the fixed page size of the media, if any, are published by the Print Manager and are used by the application.

User Interface Manager 910 (Fig. 221)

The User Interface Manager is illustrated in more detail if Fig. 225 and provides user interface management services. It consists of a Physical User Interface Manager 911, which controls status display and input hardware, and a Graphical User Interface Manager 912, which manages a virtual graphical user interface on the color display. The User Interface Manager translates virtual and physical inputs into events. Each event is placed in the event queue of the process registered for that event.

File Manager 905 (Fig. 222)

The File Manager provides file management services. It provides a unified hierarchical file system within which the file systems of all mounted volumes appear. The primary removable storage medium used in the Artcam is the ArtCards. A ArtCards is printed at high resolution with blocks of bi-level dots which directly representserror-tolerant Reed-Solomon-encoded binary data. The block structure supports append and append-rewrite in suitable read-write ArtCards devices (not initially used in Artcam). At a higher level a ArtCards can contain an extended append-rewriteable ISO9660 CD-ROM file system. The software structure of the File Manager, and the ArtCards Device Controller in particular, can be as illustrated in Fig. 226.

Network Manager 904 (Fig. 222)

The Network Manager provides "appliance" networking services across various interfaces including infra-red (IrDA) and universal serial bus (USB). This allows the Artcam to share captured images, and receive images for printing.

Clock Manager 907 (Fig. 222)

The Clock Manager provides date and time-of-day clock services. It utilises the battery-backed real-time clock embedded in the Artcam, and controls it to the extent that it automatically adjusts for clock drift, based on auto-calibration carried out when the user sets the time.

Power Management

When the system is idle it enters a quiescent power state during which only periodic scanning for input events occurs. Input events include the press of a button or the insertion of a ArtCards. As soon as an input event is detected the Artcam device re-enters an active power state. The system then handles the input event in the usual way.

Even when the system is in an active power state, the hardware associated with individual modules is typically in a quiescent power state. This reduces overall power consumption, and allows particularly draining hardware components such as the printer's paper cutting guillotine to monopolise the power source when they are operating. A camera-oriented Artcam device is, by default, in image capture mode. This means that the camera is active, and other modules, such as the printer, are quiescent. This means that when non-camera functions are initiated, the application must explicitly suspend the camera module. Other modules naturally suspend themselves when they become idle.

Watchdog Timer

The system generates a periodic high-priority watchdog timer interrupt. The interrupt handler resets the

system if it concludes that the system has not progressed since the last interrupt, i.e. that it has crashed.

Alternative Print Roll

In an alternative embodiment, there is provided a modified form of print roll which can be constructed mostly from injection moulded plastic pieces suitably snapped fitted together. The modified form of print roll has a high ink storage capacity in addition to a somewhat simplified construction. The print media onto which the image is to be printed is wrapped around a plastic sleeve former for simplified construction. The ink media reservoir has a series of air vents which are constructed so as to minimise the opportunities for the ink flow out of the air vents. Further, a rubber seal is provided for the ink outlet holes with the rubber seal being pierced on insertion of the print roll into a camera system. Further, the print roll includes a print media ejection slot and the ejection slot includes a surrounding moulded surface which provides and assists in the accurate positioning of the print media ejection slot relative to the printhead within the printing or camera system.

Turning to Fig. 227 to Fig. 231, in Fig. 227 there is illustrated a single point roll unit 1001 in an assembled form with a partial cutaway showing internal portions of the printroll. Fig. 228 and Fig. 229 illustrate left and right side exploded perspective views respectively. Fig. 230 and Fig. 231 are exploded perspective's of the internal core portion 1007 of Fig. 227 to Fig. 229.

The print roll 1001 is constructed around the internal core portion 1007 which contains an internal ink supply. Outside of the core portion 1007 is provided a former 1008 around which is wrapped a paper or film supply 1009. Around the paper supply it is constructed two cover pieces 1010, 1011 which snap together around the print roll so as to form a covering unit as illustrated in Fig. 227. The bottom cover piece 1011 includes a slot 1012 through which the output of the print media 1004 for interconnection with the camera system.

Two pinch rollers 1038, 1039 are provided to pinch the paper against a drive pinch roller 1040 so they together provide for a decurling of the paper around the roller 1040. The decurling acts to negate the strong curl that may be imparted to the paper from being stored in the form of print roll for an extended period of time. The rollers 1038, 1039 are provided to form a snap fit with end portions of the cover base portion 1077 and the roller 1040 which includes a cogged end 1043 for driving, snap fits into the upper cover piece 1010 so as to pinch the paper 1004 firmly between.

The cover pieces 1011 includes an end protuberance or lip 1042. The end lip 1042 is provided for accurately alignment of the exit hole of the paper with a corresponding printing heat platen structure within the camera system. In this way, accurate alignment or positioning of the exiting paper relative to an adjacent printhead is provided for full guidance of the paper to the printhead.

Turning now to Fig. 230 and Fig. 231, there is illustrated exploded perspectives of the internal core portion which can be formed from an injection moulded part and is based around 3 core ink cylinders having internal sponge portions 1034-1036.

At one end of the core portion there is provided a series of air breathing channels eg. 1014 - 1016. Each air breathing channel 1014 - 1016 interconnects a first hole eg. 1018 with an external contact point 1019 which is interconnected to the ambient atmosphere. The path followed by the air breathing channel eg. 1014 is preferably of a winding nature, winding back and forth. The air breathing channel is sealed by a portion of sealing tape 1020 which is placed over the end of the core portion. The surface of the sealing tape 1020 is preferably hydrophobically treated to make it highly hydrophobic and to therefore resist the entry of any fluid portions into the air breathing channels.

At a second end of the core portion 1007 there is provided a rubber sealing cap 1023 which includes three thickened portions 1024, 1025 and 1026 with each thickened portion having a series of thinned holes. For example, the portion 1024 has thinned holes 1029, 1030 and 1031. The thinned holes are arranged such that one hole from each of the separate thickened portions is arranged in a single line. For example, the thinned holes 1031, 1032 and 1033 (Fig. 230) are all arranged in a single line with each hole coming from a different thinned portion. Each of the thickened portions corresponds to a corresponding ink supply reservoir such that when the three holes are pierced, fluid communication is made with a corresponding reservoir.

An end cap unit 1044 is provided for attachment to the core portion 1007. The end cap 1044 includes an aperture 1046 for the insertion of an authentication chip 1033 in addition to a pronged adaptor (not shown) which includes three prongs which are inserted through corresponding holes (e.g., 1048), piercing a thinned portion (e.g., 1033) of seal 1023 and interconnecting to a corresponding ink chamber (e.g., 1035).

Also inserted in the end portion 1044 is an authentication chip 1033, the authentication chip being provided to authenticate access of the print roll to the camera system. This core portion is therefore divided into three separate chambers with each containing a separate color of ink and internal sponge. Each chamber includes an ink outlet in a first end and an air breathing hole in the second end. A cover of the sealing tape 1020 is provided for covering the air breathing channels and the rubber seal 1023 is provided for sealing the second end of the ink chamber.

The internal ink chamber sponges and the hydrophobic channel allow the print roll to be utilized in a mobile environment and with many different orientations. Further, the sponge can itself be hydrophobically treated so as to force the ink out of the core portion in an orderly manner.

A series of ribs (e.g., 1027) can be provided on the surface of the core portion so as to allow for minimal frictional contact between the core portion 1007 and the printroll former 1008.

Most of the portions of the print roll can be constructed from ejection moulded plastic and the print roll includes a high internal ink storage capacity. The simplified construction also includes a paper decurling mechanism in addition to ink chamber air vents which provide for minimal leaking. The rubber seal provides for effective communication with an ink supply chambers so as to provide for high operational capabilities.

Artcards can, of course, be used in many other environments. For example ArtCards can be used in both embedded and personal computer (PC) applications, providing a user-friendly interface to large amounts of data or configuration information.

This leads to a large number of possible applications. For example, a ArtCards reader can be attached to a PC. The applications for PCs are many and varied. The simplest application is as a low cost read-only distribution medium. Since ArtCards are printed, they provide an audit trail if used for data distribution within a company.

Further, many times a PC is used as the basis for a closed system, yet a number of configuration options may exist. Rather than rely on a complex operating system interface for users, the simple insertion of a ArtCards into the ArtCards reader can provide all the configuration requirements.

While the back side of a ArtCards has the same visual appearance regardless of the application (since it stores the data), the front of a ArtCards is application dependent. It must make sense to the user in the context of the application.

A further application of the ArtCards concept, hereinafter called "BizCard" is to store company information on business cards. BizCard is a new concept in company information. The front side of a bizCard, as illustrated in Fig. 232, looks and functions exactly as today's normal business card. It includes a photograph and contact information,

with as many varied card styles as there are business cards. However, the back of each bizCard contains a printed array of black and white dots that holds 1 - 2 megabytes of data about the company. The result is similar to having the storage of a 3.5" disk attached to each business card. The information could be company information, specific product sheets, web-site pointers, e-mail addresses, a resume ..... in short, whatever the bizCard holder wants it to. BizCards can be read by any ArtCards reader such as an attached PC card reader, which can be connected to a standard PC by a USB port. BizCards can also be displayed as documents on specific embedded devices. In the case of a PC, a user simply inserts the bizCard into their reader. The bizCard is then preferably navigated just like a web-site using a regular web browser.

Simply by containing the owner's photograph and digital signature as well as a pointer to the company's public key, each bizCard can be used to electronically verify that the person is in fact who they claim to be and does actually work for the specified company. In addition by pointing to the company's public key, a bizCard permits simple initiation of secure communications.

A further application, hereinafter known as "TourCard" is an application of the ArtCards which contains information for tourists and visitors to a city. When a tourCard is inserted into the ArtCards book reader, information can be in the form of:

* Maps
* Public Transport Timetables
* Places of Interest
* Local history
* Events and Exhibitions
* Restaurant locations
* Shopping Centres

TourCard is a low cost alternative to tourist brochures, guide books and street directories. With a manufacturing cost of just one cent per card, tourCards could be distributed at tourist information centres, hotels and tourist attractions, at a minimum cost, or free if sponsored by advertising. The portability of the bookreader makes it the perfect solution for tourists. TourCards can also be used at information kiosk's, where a computer equipped with the ArtCards reader can decode the information encoded into the tourCard on any web browser.

It is interactivity of the bookreader that makes the tourCard so versatile. For example, Hypertext links contained on the map can be selected to show historical narratives of the feature buildings. In this way the tourist can embark on a guided tour of the city, with relevant transportation routes and timetables available at any time. The tourCard eliminates the need for separate maps, guide books, timetables and restaurant guides and creates a simple solution for the independent traveller.

Of course, many other utilizations of the data cards are possible. For example, newspapers, study guides, pop group cards, baseball cards, timetables, music data files, product parts, advertising, TV guides, movie guides, trade show information, tear off cards in magazines, recipes, classified ads, medical information, programmes and software, horse racing form guides, electronic forms, annual reports, restaurant, hotel and vacation guides, translation programmes, golf course information, news broadcast, comics, weather details etc.

**Artcard Extra uses**

ART Descriptions

Art04 Description

In a further refinement, the Artcam device is suitably modified so as to equip it with a microphone device and associated recording technologies. When a picture is taken, the opportunity is provided to record either the surrounding sound environment or a message associated with the image. The recorded audio is then printed on the back of the output photograph in an encoded format, the encoding preferably being of a highly resilient form. The recorded audio provides a permanent audio record associated with the corresponding photograph. Subsequently, a playback apparatus is provided for scanning the encoded audio and decoding this information.

Turning now to Fig. 233, there is illustrated, in schematic form a further refinement 1601 which includes the Artcam arrangement wherein an image 1602 is sensed via a CCD sensor 1603 and forwarded to an Artcam central processor 1604 which includes significant computational resources. The Artcam central processor 1604 can store the image in memory 1605 which preferably comprises a high speed RAMBUS (Trade Mark) interfaced memory. The Artcam central processor 1604 is also responsible for controlling the operation of a printhead 1606 for the printing out of full color photographs, eg. 1607, so as to provide for instant images on demand.

In a further refinement, the camera arrangement 1601 is also supplied with a sound chip 1610 which interfaces via RAMBUS 1611 to memory 1605 under the control of the ACP processor 1604. The sound chip 1610 can be of a standard or specialised form and can, for example, comprise a DSP processor that takes an analogue input 1612 from a sound microphone 1613. Alternatively, with increasing chip complexities (Moore's Law), the functionality of sound chip 1610 can be incorporated onto the ACP chip 1604 which preferably comprises a leading edge CMOS type integrated circuit chip. It will be readily evident that many other types of arrangements can be provided which fall within the scope of the present invention.

The sound chip 1610 converts the analogue input 1612 to a corresponding digital form and forwards it for storage in memory 1605. The recording process can be activated by means of the depressing of a button (not shown) on the camera device, the button being under the control of the ACP processor 1604 otherwise it can be substantially automatic when taking a photo. The recorded data is stored in the memory 1605.

Turning now to Fig. 234, the camera arrangement preferably includes a printer device 1606 which includes two printheads 1615, 1616 with the first printhead 1615 being utilized to print an image on print media 1617 and a second printhead 1616 being utilized to print information on the back of print output media 1617.

Turning now to Fig. 235, there is illustrated an example output on the back of photo media 1617, the output being printed by printhead 1616 of Fig. 234. The information can include location, date and time data 1620 with the location data being provided by means of keyboard input or, alternatively, through the utilization of attached positioning systems such as GPS or the like. The information 1620 is presented in a viewable form for a user to maintain a record of the printed image. Importantly, on the back of the image 1617 is also printed an encoded form 1622 of the audio information. The format of the encoding can be many and various, however, preferably the encoding is provided in a highly fault tolerant manner so as to tolerate scratches, grime, writing, wear, rotation, fading etc.. The portion of the data 1625 is shown in schematic form in Fig. 236 and the data comprises a array of dots as printed by the printhead 1616 of Fig. 234. The disclosed methodologies provide for the storage of about 2.5 Megabyte of arbitrary data on the back of a photo which, in this particular instance, can

comprise sound data. The encoding format relates heavily upon utilization of Reed-Solomon encoding of the data to provide for a high degree of fault tolerance. Further, as indicated in Fig. 236, a high frequency "checkerboard" pattern is described as being added to the data so as to assist in sensing of an image. The portion of the data 1625 is shown in schematic form in Fig. 236 as the data will comprise a array of dots as printed by the printhead 1616 of Fig. 234.

When it is desired to "play back" the recorded audio, the photo 1617 is passed through a reader device 1626 which includes pinch rollers for pinch rolling the photo 1617 passing a linear CCD sensor device 1627.

Referring now to Fig. 238, there is illustrated a schematic form illustrating the operation of the audio reader device 1626 of Fig. 237. The linear CCD sensor 1627 is interconnected to a second Artcam central processor 1628 which is suitably adapted to read and decode the data stored on the back of the photograph. The decoded audio information is stored in memory 1632 for playback via a sound processing chip 1633 on speaker 1629. The sound processing chip 1633 can operate under the control of the ACP decoder 1628 which in turn operates under the control of various user input controls 1633 which can include volume controls, rewind, play and fast forward controls etc.

Importantly, the CCD linear sensor 1627 and the ACP decoder 1628 can implement the reading process as if the information were printed on the back of an Artcard as previously described.

It can be seen from the foregoing description of a further refinement that there is provided a system for the automatic recording of audio associated with an output image so as to provide an audio record associated with a photograph. There is also disclosed a audio reader system for reading an image output on the back of a photograph.

Art06 Description

In a further refinement the aforementioned Artcam includes an apparatus for the automatic detection of a cameras orientation which is subsequently utilized in the production of digital images. The Artcam is then suitably programmed utilizing the orientation information in image production.

A number of examples of utilization of orientation information will be now be described with reference to the accompanying drawings. Turning to Fig. 239, there is illustrated a first portrait photograph 1801 wherein an area 1802 is provided for the overlaying of information relevant to the portrait 1801, such as date and location information entered via the camera system. Further, it is envisaged that the portrait 1801 may be subjected to image processing techniques which distort certain visual portions of the image such as the face etc. These distortions rely upon detection of the orientation of the camera taking the photograph 1801. There is also shown in Fig. 239, a second portrait mode image 1803 wherein the camera has been utilized in a rotated portrait mode. Unfortunately if the image data 1802 of the landscape mode image 1801 is placed in the same position, it will appear in the area 1805 in the final image which is an undesirable position. Ideally, the image specific information should appear in a place 1806 when the image taken in a portrait mode. Further, image processing effects which are applied to the image in an orthogonal, anisotropic, image content specific manner will be ineffectual when the image is rotated.

The orientation sensor 1846 senses an orientation which is converted by the ACP 1831 into a corresponding digital orientation angle. The area image sensor 1802 senses the image taken by the camera which is forwarded to the ACP 1831 and stored within memory storage device 1833. Next, the sensed orientation is utilized to determine whether a portrait or landscape mode image has been taken. The sensed image can be scanned in from memory store

1833 and rotated before again being outputted to memory store 1833. The final result being a rotated image 1808. Alternatively, other techniques for utilization of the sensed orientation information are possible. For example, the sensed image stored within memory 1833 can have stored associated with it, the orientation information utilizing the traditional rotational matrix system mapping a world coordinate system and local coordinate system mapping. Subsequently, computer graphic algorithms wishing to process the information can utilize the coordinate system during processing so as to operate on the image in a rotated orientation. Where position sensitive information eg. 1806 of Fig. 239 is to be incorporated in the image, this text can be positioned in the correct orientation. Further, the orientation of the camera can be utilized in any image specific algorithmic techniques applied to the image such as face detection. The utilization of the orientation information substantially simplifies any face detection algorithm as the face orientations will be approximately known.

The orientation sensor can comprise a simple and inexpensive gravity switch, such as a mercury switch, or can be a more complex miniature device such as a micro machined silicon accelerometer.

Art07 Description

Futher, other forms of print rolls can be constructed. Turning now to Fig. 241, there is illustrated schematically an alternative arrangement of the print roll as utilized in a modified version of the aforementioned apparatus. Importantly, the disposable print roll, e.g. 1910, includes an extended leaf portion 1911 which includes a sponge portion 1912 which allows for the collection of ink therein. Hence, in operation, the sponge portion 1912 can be placed below the print-head so as to allow for printing by the print-head to the very extreme of the film or paper passed across the print-head.

The sponge portion 1912 forms part of the disposable print roll 1910 and hence, is taken out of the camera when the print roll is finished, thereby removing excess waste ink from the print-head.

Art08 Description

In a further embodiment, autofocus is achieved by processing of a CCD data stream to ensure maximum contrast. Techniques for determining a focus position based on a CCD data stream are known. For example, reference is made to "The Encyclopedia of Photography" editors Leslie Stroebel and Richard Zakia, published 1993 by Butterworth-Heinemann and "Applied Photographic Optics" by London & Boston, Focal Press, 1988. These techniques primarily rely on measurements of contrast between adjacent pixels over portions of an input image. The image is initally processed by the ACP in order to determine a correct autofocus setting.

This autofocus information is then utilized by the ACP in certain modes, for example, when attempting to locate faces within the image, as a guide to the likely size of any face within the image, thereby simplifying the face location process.

Turning now to Fig. 242, there is illustrated 2001 an example of the method utilized to determine likely image characteristics for examination by a face detection algorithm.

Various images eg. 2002, 2003 and 2004 are imaged by the camera device. As a by product of the operation of the auto-focusing the details of the focusing settings are stored by the ACP. Additionally, a current position of the zoom motor is also utilized 2006. Both of these settings are determined by the ACP. Subsequently, the ACP applies any analysis techniques 2008 to the detected values before producing an output 2009 having a magnitude corresponding to the likely depth location of objects of interest within the image.

Next, the depth value is utilized in a face detection algorithm 2010 running on the ACP 2031 in addition to the

inputted sensed image 2011 so as to locate objects within the image. A close range value 2009 indicates a high probability of a portrait image, a medium range indicates a high probability of a group photograph and a further range indicates a higher probability of a landscape image. This probability information can be utilized as an aid for the face detection algorithm and also can be utilized for selecting between various parameters when producing "painting" effects within the image or painting the image with clip arts or the like, with different techniques or clip arts being applied depending on the distance to an object.

Art09 Description

In a further embodiment, the Artcam device can be modified so as to include an eye position sensor which senses a current eye position. The sensed eye position information is utilized to process the digital image taken by the camera so as to produce modifications, transformations etc. in accordance with the sensed eye position.

The construction of eye position sensors is known to those skilled in the art and is utilized within a number of manufacture's cameras. In particular, within those of Canon Inc. Eye position sensors may rely on the projection of an infra red beam from the viewfinder into the viewer's eye and a reflection detected and utilized to determine a likely eye position.

Turning now to Fig. 243, the eye position information 2110 and the image 2111 are stored in the memory of the Artcam and are then processed 2112 by the ACP to output a processed image 2113 for printing out as a photo via a print head. The form of image processing 2112 can be highly variable provided it is dependant on the eye position information 2110. For example, in a first form of image processing, a face detection algorithm is applied to the image 2111 so as to detect the position of faces within an image and to apply various graphical objects, for example, speech bubbles in a particular offset relationship to the face. An example of such process is illustrated in Fig. 244 wherein, a first image 2115 is shown of three persons. After application of the face detection algorithm, three faces 2116, 2117 and 2118 are detected. The eye position information is then utilized to select that face which is closest to an estimated eye view within the frame. In a first example, the speech bubble is place relative to the head 2116. In a second example 2120, the speech bubble is placed relative to the head 2117 and in a third example 2121, the speech bubble is placed relative to the head 2118. Hence, an art card can be provided containing an encoded form of speech bubble application algorithm and the image processed so as to place the speech bubble text above a pre-determined face within the image.

It will be readily apparent that the eye position information could be utilized to process the image 2111 in a multitude of different ways. This can include applying regions specific morphs to faces and objects, applying focusing effects in a regional or specific manner. Further, the image processing involved can include applying artistic renderings of an image and this can include applying an artistic paint brushing technique. The artistic brushing methods can be applied in a region specific manner in accordance with the eye position information 2110. The final processed image 2113 can be printed out as required. Further images can be then taken, each time detecting and utilizing a different eye position to produce a different output image.

Art10 Description

In a further refinement, the Artcam is modified to have an auto exposure sensor for determining the light level associated with the captured image. This auto exposure sensor is utilized to process the image in accordance with the set light value so as to enhance portions of the image.

Preferably, the area image sensor includes a means for determining the light conditions when capturing an image. The area image sensor adjusts the dynamic range of values captured by the CCD in accordance with the detected level sensor. The captured image is transferred to the Artcam central processor and stored in the memory store. Intensity information, as determined by the area image sensor, is also forwarded top the ACP. This information is utilized by the Artcam central processor to manipulate the stored image to enhance certain effects.

Turning now to Fig. 245, the auto exposure setting information 2201 is utilized in conjunction with the stored image 2202 to process the image by utilizing the ACP. The processed image is returned to the memory store for later printing out 2204 on the output printer.

A number of processing steps can be undertaken in accordance with the determined light conditions. Where the auto exposure setting 2201 indicates that the image was taken in a low light condition, the image pixel colors are selectively re-mapped so as to make the image colors stronger, deeper and richer.

Where the auto exposure information Indicates that highlight conditions were present when the image was taken, the image colors can be processed to make them brighter and more saturated. The re-coloring of the image can be undertaken by conversion of the image to a hue-saturation-value (HSV) format and an alteration of pixel values in accordance with requirements. The pixel values can then be output converted to the required output color format of printing.

Of course, many different re-coloring techniques may be utilized. Preferably, the techniques are clearly illustrated on the pre-requisite artcard inserted into the reader. Alternatively, the image processing algorithms can be automatically applied and hard-wired into the camera for utilization in certain conditions.

Alternatively, the Artcard inserted could have a number of manipulations applied to the image which are specific to the auto-exposure setting. For example, clip arts containing candles etc could be inserted in a dark image and large suns inserted in bright images.

Art11 Description

One important form of processing is the removal of "red-eye" effects that can result in captured images as result of utilization of a flash. Turning now to Fig. 246, in a further refinement, the image 2302 as originally captured by the CCD device is subject to a processing step three when a flash has been utilized so as to produce a processed output image for printing. Turning now to Fig. 247, there is illustrated in more detail, one particular image processing algorithm 2310 which can be utilized when a flash has been utilized in capturing an image by a CCD device. The algorithm is preferably only utilized when a flash was used 2311 to take the picture captured by the CCD. The purpose of the algorithm is to reduce the image effects due to the utilization of the flash. Such image effects can include the well known "red-eye" effect of individual eyes appearing red in a photographic image. Other effects such as flash reflections off reflective surfaces can also be separately processed utilizing other algorithms. The first step in eliminating red-eye effects in the images is to determine 2312 the faces within the image. The face detection process can proceed by detecting regions of contiguous color which map the hue, saturation and value of HSV of the range of human face colors under the range of normal lighting encountered after any other applied image enhancements or hue corrections. The detected regions can then be passed through various huristic tests including determining the prescience of eyes, mouth, overall shape and overlap. The huristic tests produce a result in probability of a face and where this is above a threshold, a face is determined to be located in the image.

Once a face has been determined 2312 within an image, the eyes are located 2313 within the face. Each eye

is then independently processed to determine its special range of colors 2314 so as determine whether a red-eye removal process is required. If the red-eye removal process is required, a retouching algorithm is applied to the eye area so as to reduce the red saturation whilst simultaneously not introducing any discontinuities or likely artefacts in the output image. Of course, many different techniques could be utilized including a form of gaussion degree some form of gaussion degree of alteration around a central point of the eye. Finally, the image is written 2316 in its updated form, back to the memory store 2333.

Preferably, any other retouching algorithms including remapping colors affected by the spectrum nature of the flashlight are also utilized at this time. Alternatively, the Artcard inserted could have a number of manipulations applied to the image which are specific to the flash setting. For example, clip arts containing candles, light globes etc could be inserted in an image utilizing a flash and large suns inserted in non-flash images.

Art12 Description

In a further refinement a second ink jet is provided for printing on the back of a photo. As illustrated in Fig. 248, there is shown a schematic view of the normal arrangement. The print-head 2402 is provided for printing photographs on paper 2403 which is pinched between rollers, e.g. 2404. In Fig. 249, there is illustrated an alternative arrangement 2410 in accordance with the this embodiment which includes two ink jet print-heads 2402, 2411 with the print-head 2411 being provided for printing on the back of the photograph 2403 important information of relevance to the photograph. For example, the time and date that the photograph was taken can be printed on the back by ink jet print-head 2411. Additionally, the Artcam device could be provided with a global positioning sensor which could be interrogated by the artcam central processor and utilized to determine a location that the photograph taken at. This additional information could also be printed on the back of the photo output 2403.

Art13 Description

The print roll is provided on a tightly wound former. Unfortunately, the print media characteristics may be such that an excessive amount of curl is provided in any photograph output by the artcam system as a result of the tight rolling of the print roll.

Referring to Fig. 250, there is illustrated an alternative arrangement of print roll feeding the system which includes a three roller arrangement 2511-2513, which provides a tight "anti-curl" curl to the print medium 2514 which is exiting the print roll. The three rollers 2511-2513 pinch the media 2514 and provide the anti-curl to the paper before it is forwarded to the print-head 2516 via pinch rollers, e.g. 2517. The three rollers 2511-2513 can be provided within the print roll casing or alternatively within the artcam device.

Art15 Description

Referring now to Fig. 251, in a further refinement, it is desired to create a photographic image to which allows a viewer 2703 to view stereoscopical or pseudo three-dimensional type effects. These effects can be viewed by simultaneously recording two images close together and presenting one image to the viewer's right eye 2704 and a second image to the viewer's left eye 2705. By recording the image for the left and right eye and then presenting a surface which allows the left eye to view the left stereoscopic image and the right eye to view the right stereoscopic image, a 3-D stereoscopic effect will be produced.

In a further refinement, the photo or stereoscopic image 2702 can be constructed, as will become more apparent hereinafter, by means of a series of lenticular transparent columns which image the left and right eye images.

Turning now to Fig. 252, there is illustrated a cross-sectional view of part of the surface of the photographic paper 2702. As is illustrated in cross-sectional view 2710, the photographic paper consists of a series of lenticular columns 2711 designed to separate out the left and right stereoscopic view. The image is printed on the bottom side 2712 of the transparent photographic paper 2710.

It is assumed that the pitch 2713 of a single pixel is 80 mm and the pitch of the dot spacing is at 20 mm 2714. Hence, four dots 2714-2717 are provided per pixel. Two dots 2714-2715 are provided for the left stereoscopic view and two dotsx2716- 2717 are provided for the right stereoscopic view. The right eye viewing the photographic paper will image 2720,2721 the right hand pixel dots 2716, 2717 and the left eye will image 2723,2724 the left hand dots 2725,2726. Therefore, the two images are presented separately to each eye and a stereo photographic effect results. The stereo photographic effect primarily resulting from the lenticular profile of each column e.g. 2711.

Turning now to Fig. 253, there is illustrated a perspective view of the underside of a portion of paper 2710. The illustration of Fig. 253 includes construction lines so as to illustrate the dot pitch of each dot 2730 in addition to the pixel pitch, which includes four dots 2731-2734. As illustrated in Fig. 253, four dots can be provided for each pixel with two dots 2731-2732 being provided for the right hand stereographic view and two dots 2733,2734 being provided for the left hand stereographic view. The dots e.g. 2730 are laid down on the print media 2710 so that, when the media is reversed, a correct stereoscopic view results.

Turning now to Fig. 254, one form of imaging a stereoscopic image will now be discussed. Two images are preferably imaged by CCD couplers 2740,2741 which are located in a portable camera device. Each CCD device 2740,2741 images a separate image which is to be combined via the stereo processing unit 2742 in the left and right interleaved manner as illustrated in Fig. 253. Upon correctly interleaving the image by stereo processing unit 2742, the image is sent to print engine 2743 and to printout 2744 which prints the image on the top side 2746 of photographic paper 2710.

The current position of the photographic paper 2710 is detected by positioning unit 2750 with the current position being fed back to print engine 2743 for the control of print head 2744. The position is determined by means of an LED type device 2780 imaging the lenticular surface of the print media 2710 with the periodic variation in intensity being measured by a photo conductor 2781 on the opposite side of the print media 2710, thereby giving an accurate measure of the motion of the paper 2710.

Turning to Fig. 255, the operation of the positioning unit indicated by the broken line 2750 will now be further explained. The photographic paper 2710 is pinched between rollers 2751,2752, the roller 2752 is constructed to have a mating surface to the lenticular surface 2754 of the print media 2710. As the lenticular period of the print media 2710 is of the order of 2780 mm, the mating surface of roller 2752 is shown in an exaggerated form, as is the lenticular surface of printer paper 2710. The fine period of the lenticular print media means that roller 2752 can be utilized with other forms of print media as well.

The surface of roller 2752 insures that the print media 2710 maintains a constant spatial relationship with the roller 2752. Hence, the print rollers 2751,2752 operate under the control of print engine 2743 as monitored by the positioning unit 2750 so as to turn roller 2752 in accordance with requirements and thereby draw print media 2710 past print head 2744. As a result, alignment between the position of dots printed by print head 2744 and the lenticular columns is maintained.

The stereographic image that can then be printed by print head 2744 as the print media 2710 is drawn past the print head. Preferably, the print head 2744 is a full image width print head.

Of course, other means of determining the current position of print media 2710 relative to the print head 2744 are possible.

Turning now to Fig. 256, a camera device can be provided in accordance with the principles of the proposed embodiment by coupling a printer roll 2760 having lenticular paper 2761 of a suitable form and a corresponding ink supply (not shown). The paper 2761 is ejected from the print roller 2760 by means of pinch rollers 2762,2763 to printing device 2764 which includes further pinch rollers 2765,2766, a cutter 2767, platen 2769, pinch rollers 2751,2752 and print head 2744. A stereographic image is then printed by print head 2744 and ejected 2770 after the paper 2761 has been cut by cutter 2767, the output 2770 being of a transparent form.

Turning now to Fig. 257, upon ejection of an image 2770, it can be adhered to a plain white surface 2771. This can be achieved by utilizing an adhesive surface 2771 to stick the transparency 2770. Subsequently, the image can be viewed as illustrated in Fig. 251 through the lenticular lens system so as to produce left and right stereographic images and thereby produce 3-dimensional effects in a taken image. Ideally, the camera system that is depicted in Fig. 256 can be of a portable form such that it can be conveniently carried to a site where pictures are desired to be taken and stereoscopic photographic pictures immediately taken.

Art17 Description**

Art21 Description

In a further refinement, the imaging media to be stored in a tightly rolled form is processed to have a anisotropic ribbed structure which allows the print media to be suitable for carriage in a roll form but further allows for the anisotropic rib structure to be utilized when viewing images printed on the printed media so as to reduce the extent of curling of the printed media.

Referring now to Fig. 259, there is illustrated 3310 print media 3311 as treated in accordance with the principles of this refinement. The print media 3311 is preferably constructed of a plastic planar film although other forms of film 3311 would be suitable. The print media 3311 is pre-treated to have a polymer rib-like structure 3312 shown in Fig. 259 in an exaggerated form with the actual column pitch being approximately 200 m. Hence, one surface of the print media 3311 is treated so as to have a series of columns eg. 3312 running up and down the length of the print media 3311, the cross-section of the column 3312 being illustrated in an enlarged form in Fig. .

The advantage of utilizing the series of columns 3312 is evident when a force 3313 is applied to a portion of the surface area of the print media 3311. The column 3312, although allowing the print media 3311 to be rolled tightly, resists any rolling of the print media 3311 in the direction 3316. Hence, any force applied 3313 is likely to be transmitted both in directions 3315 and 3316. The anisotropic nature of the media 3311 will result in the ribs 3312 acting to provide support both in directions 3315 and 3316 thereby limiting the curl of media 3311. The resistance primary being due to the resistance of ribs 3312 from any bending in the direction 3316. The anisotropic strength of the material also allows it to be stored on a roll with the greater strength axis being along the central axis of the roll.

Referring now to Fig. 260, the image on media 3311 can then be viewed, with reduced effects due to curling, by merely holding the image 3311 in a users hands 3320, 3321 and applying subtle pressure at points 3324,

3325. The application of pressure at points 3324, 3325 is transmitted throughout the media 3311 thereby providing a flat viewing surface with limited curling.

Of course, the anisotropic media can be produced utilizing a number of techniques where the media comprises a plastic type material. For example, it could be produced by methods of extrusion. Alternatively, other techniques can be utilized. For example, one form of production 3340 is illustrated schematically in Fig. 261 and relies upon the media 3341 being pressed between rollers 3342, 3343. The roller 3343 having a flat surface with the roller 3342 having a serrated surface 3345 as illustrated in Fig. 261a. The dimensions of the serration are exaggerated in Fig. 261 for illustrative purposes.

Where the media is unsuitable for utilization in such a process 3340, the anisotropic media can be provided by utilizing two film surfaces joined together. Such a process 3350 is illustrated in Fig. 262 wherein a first surface 3351 upon which it is desired to print an image on is mated with a second surface 3352 by means of glue, heat fusion etc. as required.

Further, other forms of manufacture of the print media possible. For example, referring to Fig. 263 and Fig. 264, the construction of print media 3360 is shown in cross-section and could be constructed from a first fibrous material 3361 which can comprise strong polymer based fibres constructed from annealed polyethylene naphthalate drawn into a fibrous form. A second heat flowable polymer 3362 which can comprise polyethylene or the like is utilized to provide the "paper" base for the carriage of ink/imaging chemicals. The media polymer 3362 can be compressed with the fibrous material 3361 while still in a viscous form (for example, after being heated) so that the fibre 3361 and print base 3362 form one integral unit as illustrated in Fig. 264. The media arrangement of Fig. 263 and Fig. 264 can be constructed in accordance with a number of techniques. Referring now to Fig. 265, there is illustrated a first such technique which utilizes pre-formed spools eg. 3380 of fibrous material, the number of spools and corresponding fibres 3381 being equivalent in number to the length of the desired print roll. The spools 3380 are ranged such that corresponding fibres 3381 are fed over a first roller 3382. The fibres 3381 are then drawn past a print media application unit 3384 which applies the layer of viscous heated print media to the fibres. Subsequently, rollers 3385, 3386 apply the necessary pressure to the fibres and print media so as to fuse the two together and flatten the surface of the print media so as to form a final anisotropic surface 3388. Subsequently, the surface 3388 is cut across its width and rolled to form anisotropic printing media as hereinbefore described.

Of course, alternative forms of creating the fibrous material on demand are possible. For example, in Fig. 266, there is illustrated one form of creating an extruded fibrous material by means of drawing fibre eg. 3391 from a vat 3390, the arrangement of Fig. 266 designed to replace the need for spools 3380 of Fig. 265.

<u>Art24 Description</u>

In a further refinement there is described an algorithm for utilization on the Artcam system, which will automatically convert a photographic image into a "painted" rendition of that image which replaces groups of pixels in the input image with "brush strokes" in the output image. The algorithm works by automatically detecting dominant edges and propagating the edge direction information into flat areas of the image so that brush strokes can be oriented in such a way as to approximate the van Gogh style. The algorithm is suitable for implementation on the aforementioned Artcam device.

Turning initially to Fig. 267, the algorithm comprises a number of steps 3601. These steps include an initial step of filtering the image to detect its edges 3602. Next, the edges are thresholded or "skeletonised" 3604 before

being processed 3605 to determine the final edges 3606. Bézier curves are then fitted to the edges. Next, the curves are offset 3607 and brush strokes are placed on final image 3608. The process 3607 and 3608 is iterated until such time as the image is substantially covered by brush strokes. Subsequently, final "touching up" 3609 of the image is performed.

Turning now to describe each step in more detail. In the first step 3602 of filtering to detect edges, a Sobel 3 x 3 filter having co-efficient sets 3612 and 3613 as illustrated in Fig. 268 can be applied to the image. The Sobel filter is a well known filter utilized in digital image processing and its properties are fully discussed in the standard text "Digital Image Processing" by Gonzalez and Woods published 1992 by the Addison - Wesley publishing company of Reading, Massachusetts at pages 197-201. The Sobel derivative filter can be applied by either converting the image to greyscale before filtering or filtering each of the color channels of an image separately and taking the maximum. The result of Sobel filtering is the production of a greyscale image indicating the per-pixel edge strength of the image.

Next, the resultant per-pixel edge strength image is thresholded 3603 so as to produce a corresponding thresholded binary image. The threshold value can be varied however, a value of 50% of the maximum intensity value is suitable. For each pixel in the edge strength image the pixel is compared with the threshold and if it is greater than the threshold a "one" is output and if it is less than the threshold a "zero" is output. The result of this process is to produce a threshold edge map.

Next, the thresholded edge map is "skeletonised" at step 3604 of Fig. 267. The process for skeletonising an image is fully set out in the aforementioned reference text at pages 491-494 and in other standard texts. The process of skeletonisation produces a "thinned" skeletonised edge map maintaining a substantial number of characteristics of the thresholded edge map.

In a next step the edges of the skeletonised edge map are determined to yield a data structure which comprises a list of further lists of points within the image. Preferably, only edges having a length greater than a predetermined minimum are retained in the list.

As the skeletonised image contains only single-pixel-width edges, possibly with multiple branches, the following algorithm expressed as a C++ code fragment sets out one method of determining or identifying the points which belong to each contiguous edge in the skeletonised image. It breaks branching edges into separate edges, and chooses to continue along the edge in the direction which minimises the curvature of each branch - ie. at a branch-point it favours following the branch which induces the least curvature. The code is as follows:

```
void
FollowEdges
(
        Image& image,
        int minimumEdgeLength,
        PointListList& pointListList
)
(
        pointListList.Erase();
        for (int row = 0; row < image.Height(); row++)
        {
```

```
            for (int col = 0; col < image.Width(); col++)
            {
                If (image[row][col] > 0)
                {
                    PointList pointList;

                    // append the starting point to the point list,
                    // and clear it so we don't find it again
                    pointList.Append(Point(col, row));
                    image[row][col] = 0;

                    // follow the edge from the starting point to its beginning
                    FollowEdge(row, col, image, pointList);

                    // reverse the order of the points accumulated so far,
                    // and follow the edge from the starting point to its end
                    pointList.Reverse();
                    FollowEdge(row, col, image, pointList);

                    // keep the point list only if it's long enough
                    if (pointList.Size() >= minimumEdgeLength)
                    pointListList.Append(pointList);
                }
            }
        }
}

// table of row and column offsets to eight surrounding neighbours
// (indexed anti-clockwise, starting east)
static int offsetTable[8][2] =
{
        {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -1}, {1, -1}, {1, 0}, {1,1}
};

// table of preferred neighbour checking orders for given direction
// (indexed anti-clockwise, starting east favouring non diagnals)
static int nextDirTable[8][8] =
{
```

```
    {0,  2,  6,  1,  7,  3,  4,  5),
    {2,  0,  1,  3,  7,  4,  5,  6),
    {2,  4,  0,  3,  1,  5,  6,  7},
    {4,  2,  3,  5,  1,  6,  7,  0),
    {4,  6,  2,  5,  3,  7,  0,  1},
    {6,  4,  5,  7,  3,  0,  1,  2),
    {6,  0,  4,  7,  5,  1,  2,  3},
    {0,  6,  7,  1,  5,  2,  3,  4),
};

void
FollowEdge
(
    int row,
    int col,
    Image& image,
    PointList& pointList
)
{
    Vector edgeHistory[EDGE_HISTORY_SIZE];
    int historyIndex = 0;

    for (;;)
    {
        // table of pre-computed
        // compute tangent estimate from edge history
        Vector tangent;
        for (int i = 0; i < EDGE_HISTORY_SIZE; i++)
            tangent += edgeHistory[i];

        // determine tangent angle and quantize to eight directions
        // (direction zero corresponds to the range -PI/8 to +PI/8, i.e east)
        double realAngle = tangent.Angle();
        int angle = (int) ((realAngle * 255) / (2 * PI) + 0.5);
            int dir = ((angle - 16 + 256) % 256) / 32;

            // try surrounding pixels, fanning out from preferred
            // (i.e. edge) direction
            int* pNextDir = nextDirTable[dir];
```

```
        bool bFound = false;

    for (i = 0; i < 8; i++)
    {
            // determine row and column offset for current direction
            int rowOffset = offsetTable[dir][0];
            int colOffset = offsetTable[dir][1];

            // done testing neighbours if edge pixel found

            if (image [row + rowOffset] [col + colOffset] > 0)
            {
                // determine edge pixel address
                Point oldPoint (col, row);
                row += rowOffset;
                col += colOffset;
                Point newPoint (col, row);

                // update edge tangent history
                tangent = newPoint - oldPoint;
                tangent.Normalize();
                edgeHistory[histroyIndex] = tangent;
                historyIndex = (historyIndex + 1) % EDGE_HISTORY_SIZE;

                // append edge pixel to point list
                pointList.Append(newPoint);

                // clear edge pixel, so we don't find it again
                image[row][col] = 0;
                bFound = true;
                break;
            }
            // determine next direction to try
            dir = pNextDir[i];
        }

        // done following edge if no edge pixel found
        if (!bFound)
            break;
```

```
        }
}
```

The result of utilizing this algorithmic component on the skeletonised edgemap is to produce a list of edges having at least a predetermined size. A suitable size was found to be a length of 20 pixel elements.

In the next step 3606 of Fig. 267, Bézier curves are fitted to each of the edge lists derived from step 3605. For each list of edges, a piece wise Bézier curve is fitted to the corresponding list of points. A suitable algorithm for fitting the piece wise Bézier curve is Schneider's curve fitting algorithm as set out in Schneider, P.J., "An Algorithm for Automatically Fitting Digitised Curves", in Glassner,A.S. (Ed.), Graphics Gems, Academic Press, 1990. This algorithm provides quick convergence to a good fit which aims only for geometric continuity and not parametric continuity. Schneider's algorithm is recursive, such that if the fit is poor, is sub-divides the curve at the point of maximum error and fits the curves to the two halves separately. Next an estimate of the tangent at the split point is derived using only the two points on either side of the split point. For dense point sets, this tends to amplify the local noise. An improved quality of curve fitting can be alternatively undertaken by using points further away from the split point as the basis for the tangent.

In the next steps 3607 of Fig. 267, the curves are offset from the primary curve list by half a desired "brush stroke width". The offsetting occurring on both sides of the primary curve list with the result being two curves approximately one stroke width apart from one another which run parallel to and on either side of the original primary curve.

The following algorithm is utilized to generate a piece wise Bézier curves which are approximately parallel to a specified piece wise Bézier curves and includes the steps.

i. Create an empty point list.

ii. Create and empty tangent (vector) list.

iii. Evaluate selected points on each curve segment making up the piece-wise curve and offset them by the specified offset value. Append the offset points to the point list, and their corresponding tangents to the tangent list. This process is described below with reference to Fig. 268 and Fig. 269.

iv. Fit a piece-wise Bézier curve to the resultant point list. Rather than estimating tangents during the curve-fitting process, use the exact tangents associated with the offset points.

Offset each curve segment as follows:

i. Evaluate the curve value, normalised tangent and normalised normal normalised to the size of the image for a set of evenly-spaced parameter value between (and including) 0.0 and 1.0 (eg. a spacing of 0.25).

ii. Scale the normals by the specified offset value.

iii. Construct line segments using the curve points and scaled normals.

iv. If any two line segments intersect, eliminate the point associated with one of them.

v. Append the surviving points to the point list, and append their corresponding tangents to the tangent list. Only append the point associated with parameter value 1.0 if the segment in question is the last in the piece-wise curve, otherwise it will duplicate the point associated with parameter value 0.0 of the next segment.

The process of offsetting each curve segment can proceed as following:

1. Firstly, for a set of evenly spaced parameter values on the Bézier curve between (and including) 0.0

and 1.0, for each parameter value PN (Fig. 269) the curve value 3630 a normalised tangent 3631 and normalised normal 3632 are calculated.

2. Next, the normals 3632 are scaled 3634 by a specified offset value.

3. Next a line segment from the point 3630 to a point 3636, which is at the end of the scaled normal 3634 is calculated.

4. Next, the line segment 3630, 3636 is checked against corresponding line segments for all other points on the curve eg. 3638, 3639. If any two line segments intersect, one of the points 3636 is discarded.

5. The surviving points are appended to the point list and their corresponding tangents are appended to the tangent list. The point associated with the parameter value 1.0 is appended only if the segment in question is the last in the piece-wise curve segment. Otherwise, it will duplicate the point associated with the parameter value 0.0 of the next segment.

Turning to Fig. 270, the end result of the offset of curves in accordance with step 3607 of Fig. 267 is to produce for a series of Bézier curve segments C1, C2 etc. Firstly, a series of parametrically spaced points, P1 - P5. Next, the normalisation points N1 - N5 are produced (corresponding through to point 3636 of Fig. 269), for each of the points P1 - P5. Next, the resultant piece-wise Bézier curve segment 3640 is produced by utilizing the points in 1 - N5. This process is then repeated for the opposite curve comprising the points N6 - N10 and curve 3641. This process is then repeated for each of the subsequent piece-wise curves C2 etc. The result is the two curves of 3640, 3641 being substantially parallel to one another and having a spaced apart width of approximately one brush stroke.

Next, a series of brush strokes are placed into the output image along the curves. The strokes are oriented in accordance with the curve tangent direction. Each brush stroke is defined to have a foot print which defines where it may not overlap with other brush strokes. A brush stroke may only be place along the curve if its foot print does not conflict with the foot prints already present in the output image. Any curves that do not have any brush strokes placed along them are discarded and the process of steps 3607 and 3608 are iterated in a slightly modified form until no curves are left. The slightly modified form of step 3607 is to offset the curves by one brush stroke in the outward direction rather than the half brush stroke necessary when offsetting curves from the curve C1 of Fig. 270.

It has been found by utilization of the above method that the result produced consists of a series of brush strokes which emanate from objects of interest within the image.

Subsequent to covering the image with brush strokes of a given size, further processing steps can be undertaken with smaller and smaller brush strokes and increasing derivative threshold levels so as to more accurately "brush stroke" important features in the image. Such a technique is similar to that used by van Gogh in certain portions of his images where details are required.

Art26 Description

It would be further desirable that, for any image, for example a photograph, to be able to determine regions of interest within the photograph, i.e., the location of faces, etc. Once a face has been located in an image, it then becomes a simple matter to process the image to achieve a number of novel effects such as the placement of speech bubbles relative to the facial region as illustrated in Fig. 271. Other effects such as morphing of the face, placement of objects relative to the face such as glasses, etc. can also be carried out once the facial region has been detected.

As noted previously, the detection of regions in an image can be a precursor to region-specific processing, such as the application of region-specific artistic effects, for example painting the background sky with broad brush

strokes but painting the foreground face with detailed brush strokes, the detection of features in an image, etc. Colour based region detection is extremely useful as a precursor to the detection of people's faces in an image where the colors detected are of a skin tone nature.

As a result of variations in lighting, an object with a particular spectral color may appear in an image with a range of colors having variable luminance and saturation. The color of an object will tend to differ more in luminance and saturation than in hue, since hue measures the dominant frequency of an object. As illustrated in Fig. 272, the method 3801 of a further refinement includes a first step, where the image captured is first converted to a color space which distinguishes hue from luminance and saturation. Two such well known color spaces are the HLS color space and the HSV color space. The various color spaces including the aforementioned color spaces are fully described in the standard texts such as Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., Computer Graphics, Principles and Practice, Second Edition, Addison-Wesley, 1990 at pages 584-599.

The next step is to input a number of input color seeds 3804. Each seed 3805 is defined by the following:

(1) A seed color range of color values;

(2) A local color difference limit;

(3) A global difference limit.

The next step 3808 is to process the seeds to construct a bitmap.

The final step 3807 of the method of Fig. 271 comprises outputting a bitmap having the same dimensions as the input image but which indicates which pixels in the input image belong to a color region defined by the list of seeds 3805. The method 3801 comprises handling each seed in a separate pass 3808 and combining the output bitmap of each pass with the overall output bitmap. Any set bit in the individual bitmap is written to the overall bitmap.

Turning now to Fig. 272, there is illustrated the step 3808 of Fig. 271 in more detail. In a first step 3820, a seed bitmap is determined for all the colors whose seed lies in the seed color range as defined for the seed in question. In a next step 3821, for each seed in the seed bitmap is utilized as a seed location for a "seed fill" algorithm which detects 4-connected or alternatively 8-connected regions in the input image which are connected to the seed location. The seed fill algorithm flags the pixels which belong to the region in a seed output bitmap. The seeds flagged in the seed output bitmap at the step 3821 are also cleared from the seed bitmap created at the step 3820 so they are not used as subsequent starting points for subsequent seed fills.

Once all the seeds have been utilized for the seed filling process, the seed output bitmap is combined with the overall output bitmap for all of the list of seeds.

The seed fill algorithm can be of a standard type such as those disclosed in the aforementioned reference at pages 979-986. However, the seed fill algorithm has a particular pixel membership test.

A pixel is a member of the current region if its color differs from the previous pixel's color by no more than the local color difference limit, and its color differs from the seed color by no more of an image than the global color difference limit. The color limits are enforced separately for each color component. Since spans are detected or processed from left-to-right, the previous pixel is the pixel to the left of the current pixel. The color of the previous pixel at the start of a new span is taken to be its parent span's average color. The seed color is taken to be the middle color of the seed color range.

The use of a list of seeds, rather than a single seed, is motivated by wanting to identify a class of regions which may have a set of well-known but distinct color signatures, such as faces.

The use of the combination of the local color difference limit and the global color difference limit is motivated by the wanting to detect regions as completely as possible while not crossing discontinuities in the image. A region may contain a variety of colors, but these will tend to vary smoothly across the region. The global color variation in a region is in general greater than the local color variation. Imposing a local color difference limit reduces the risk of crossing discontinuities in the image. It also allows the global color difference to be less conservative, which means that region detection can be more complete even where the region becomes, for example, smoothly light or dark.

When the color space of the input image is hue-based, the hue difference limits are preferably scaled by the luminance (or value) and saturation, since for low luminance or low saturation the hue is somewhat unreliable. Thus when the hue becomes unreliable in dark or desaturated areas of a region, its effect is diminished so that those areas are still included in the detected region. This is still subject, of course, to the difference limits on the luminance and saturation.

The hue difference can also be calculated circularly, in line with the circular nature of the hue hexagon.

It has been found that utilizing the above algorithm to detect objects in images, very suitable results are often produced. For example, the above method was utilized on an image having a number of individuals and wherein the color seeds were set to be standard skin tones. The resulting regions accurately located the faces of individuals within the image.

Art27 Description

In a further refinement, as illustrated in Fig. 274, a source image 3901 is transformed into a corresponding "tiled" color image 3902, the conversion process being hereinafter described. In a first refined embodiment, each tile 3904 is placed in a regular arrangement in the output image 3902. Further, each tile 3904 derives its color from the color of a corresponding pixels 3905 in the input image 3901. The process of tiling in the output image may be done at an arbitrarily higher resolution than the input image 3901 and hence the tiling process can be used to, in effect, add detail to a low resolution image.

Of course many different tile shapes are possible. Preferably the tile shapes tiling the image can be combined into a "pattern unit" which itself, when replicated tiles the whole image. For example, referring to Fig. 2, the individual tiles 3908 and 3909 together for a pattern unit 3907 which, when replicated, will "tile" the image.

In an implementation of a further refined embodiment, each tile can be defined by an image having two channels. The first channel 3970, as illustrated in Fig. 276, defines the tile's shape and opacity. A non zero opacity area 3971 again defines those pixels which are part of the tile. The value of each pixel 3971 indicates the opacity of that pixel. Turning to Fig. 277 and Fig. 278, a second channel 3980 defines a pixel surface texture of the tile as a height field. In Fig. 278, there is illustrated a cross-section through the portion A - A' of Fig. 277 which illustrates an intensity or height peak 3991 and ramps 3992 and 3993.

The input to the algorithm is the following:

A color iamge to be reproduced as a tiling.

A tile pattern, previously describved

An output resolution

Additional optical input parameters are described below.

The output from the alforithm is a tiled rendition of the input image.

A repeating tile "pattern unit" can then be defined by:

(1) the size of the pattern unit (number of rows, number of columns);

(2) an offset to the starting point in the pattern unit for the first pixel of an image (the offset being in rows and columns);

(3) an offset between placement of successive pattern units in a vertical direction;

(4) a list of tiles which make up a pattern unit;

(5) the offsets between successive tile placements horizontally within a pattern unit.

(6) The resolution of the pattern (ie. Of the patterns' tile images).

The width of the pattern is defined to be the sum of the column offsets between successive tile placements horizontally.

All offsets are in continuous pixel coordinates (i.e. real-valued row and column offsets). This allows tiles to be placed with sub-pixel precision. This allows the resolutions of the input image, output image, and tile pattern to be independent.

The basic tiling algorithm is then embodied in the following C++ code fragment:

```
Tile
(
    Image& inputImage,
    TilePattern& pattern,
    double outputResolution,
    Image& outputImage
)
{
    // initialise output image
    double inputToOutputScale =
        outputResolution / inputImage.Resolution();
    int outputHeight = (int)ceil(inputImage.Height() * inputToOutputScale);
    int outputWidth = (int)ceil(inputImage.Width() * inputToOutputScale);
    outputImage.Initialise(outputHeight, outputWidth, outputResolution);

    // scale pattern to input resolution
    double patternToInputScale =
        inputImage.Resolution() / pattern.Resolution();
    inputPattern = pattern;
    inputPattern.Scale(patternToInputScale);

    // scale pattern to output resolution
    double patternToOutputScale =
        outputImage.Resolution() / pattern.Resolution();
    outputPattern = pattern;
    outputPattern.Scale(outputScale);
```

```
// compute output to input scale factor
double outputToInputScale =
	inputImage.Resolution() / outputImage.Resolution();

// tile image
double patternStartRow = -outputPattern.StartRow();
double patternStartCol = -outputPattern.StartCol();
for
(
	double row = patternStartRow;
	row < (double)outputImage.Height();
	row += outputPattern.RowOffset()
)
{
	double tileStartRow;
	for
	(
		double col = patternStartCol,
			int nextTile = 0,
				Tile inputTile = inputPattern.Tile(nextTile),
					Tile outputTile = outputPattern.Tile(nextTile);
		col < (double)outputImage.Width();
		col += tile.ColOffset(),
			nextTile = (nextTile + 1) % pattern.TileCount(),
				inputTile = inputPattern.Tile(nextTile),
					outputTile = outputPattern.Tile(nextTile)
	)
	{
		// reset tile start row at start of pattern
		if (nextTile == 0)
			tileStartRow = 0;

		// compute tile position in input
		double tileInputRow = row + tileStartRow;
		tileInputRow *= outputToInputScale;
		double tileInputCol = col;
		tileInputCol *= outputToInputScale;

		// compute integer and fractional position
```

```
            int intTileInputRow = (int)(tileInputRow + 0.5);
            double fracTileInputRow = tileInputRow – intTileInputRow;
            int intTileInputCol = (int)(tileInputCol + 0.5);
            double fracTileInputCol = tileInputCol – intTileInputCol;

            // scale input tile image
            Image inputTileImage = inputTile.Image();
            Translate(inputTileImage, fracInputTileRow, fracInputTileCol);

            ... determine the color of the tile from the input image
            ... i.e. inputTileImage at (intTileInputRow, intTileInputCol)

            // compute tile position in output
            double tileOutputRow = row + tileStartRow;
            double tileOutputCol = col;

            // compute integer and fractional position
            int intTileOutputRow = (int)(tileOutputRow + 0.5);
            double fracTileOutputRow = tileOutputRow – intTileOutputRow;
            int intTileOutputCol = (int)(tileOutputCol + 0.5);
            double fracTileOutputCol = tileOutputCol – intTileOutputCol;

            // scale output tile image
            Image outputTileImage = outputTile.Image();
            Translate(outputTileImage, fracOutputTileRow, fracOutputTileCol);

            ... composite the tile with the output image
            ... i.e. outputTileImage at (intTileOutputRow, intTileOutputCol)

            // update tile start row
            tileStartRow += outputTile.RowOffset();
        }

        // update pattern start col
        patternStartCol += outputPattern.ColOffset();
        while (patternStartCol > -outputPattern.StartCol())
            patternStartCol -= outputPattern.Width();
    }
}
```

As previously noted, each tile has a shape which is defined by its shape/opacity channel. The tile's shape and the tile's placement in the image together determine a set of pixels which are covered by the tile in the input image. The color of the tile in the output image is a function of this set of pixels. Colouring functions can include, but are not limited to, copying the input colors pixel-by-pixel, taking the average of the input colors, taking the median of the input colors, etc.

The colored tile may either be written directly to the output image, ignoring the tile's opacity, or may be composited with the output image according to the tile's opacity. In the latter case the output image must first be initialised to some color value, such as black, or to some arbitrary image (e.g. the input image itself). Compositing utilizing an opacity mask or channel can utilize standard compositing techniques such as those disclosed in standard articles e.g.: Thomas Porter and Tom Duff (1984) "Compositing Digital Images" SIGGRAPH Proceedings 1984 pages 253 - 259.

Each tile has a surface texture map which is defined by its texture channel. The texture channel defines a relative surface height of each pixel in the tile. This height field is later used to compute surface normals which in turn are used to determine the angles of reflection of simulated incident light. When the tile, or the image in which the tile is placed, is subject to simulated lighting, the surface texture thus becomes more apparent.

Whether the tile is written directly to the output image or is composited with the output image, the tile's texture is, in the simplest case, written directly to the output image's texture channel.

Each tile's opacity channel controls the compositing of the tile with the background. In addition, an image-sized opacity channel may also be specified which controls the compositing of the entire tiling with the background. In the simplest case the tile's opacity is scaled by the global opacity to yield the opacity used for compositing the tile (on a pixel-by-pixel basis). Turning now to Fig. 279, there is illustrated an example of the process of utilizing a global tiling opacity. In a first plot 4000 there is shown an example cross-sectional profile along a line of tiles each having a simple form of replicated opacity profile eg. 4001, 4002 etc. Next, there is illustrated a desired global opacity channel 4004 which in this example generally consists of a series of step increases in opacity 4005. In the aforementioned simple case, the tile opacity profile 4001 is scaled with the global opacity profile 4004 to produce a resultant opacity 4006 which has a series of increasing opacity areas, e.g. 4007.

Of course, other schemes of combination of the local tile opacity with a global opacity channel are possible. For example, the opacity of the pixels under each tile may be first averaged and the tile's pixel-by-pixel opacity scaled by this average opacity rather than the pixel-by-pixel global opacity, giving the tile a uniform scaled opacity.

It has been surprisingly found that, with a few simple modifications, the aforementioned tiling algorithm lends itself readily to producing painting effects. By treating the tile pattern unit as a collection of brush strokes and designing each of the tile images to look like a brush stroke, for example, with streaks from the brush hairs and greater thickness at the end of the stroke, etc., a painting effect can be achieved by setting the inter-tile and inter-pattern unit offsets so that the strokes overlap. Next, the algorithm is modified to select the strokes from the stroke collection at random, rather than iterating through the tiles in the pattern sequentially. Further, a number of texturing, positioning and coloring effects as described below can be alternatively additionally provided.

Firstly, there are typically various levels of detail in an input image which require different levels of detail when reproduced via the tiling/painting algorithm. This may be achieved by processing the image in multiple passes, utilizing a different scale stroke collection in each pass, and compositing the results of each pass with the (cumulative)

background. An image-sized detail map, consisting of a per-pixel details measure, may be used to control which parts of the image each pass processes. Each pass is given a "detail threshold" above which it processes corresponding portions of the image. Successive passes are given increasing levels of detail, and decreasing stroke collection scales. The first pass thus, for example processes the entire image, the second pass adds a certain level of details, the third pass adds still more details, and so on. This is somewhat analogous to how a human painter refined a painting by adding more and more detail. The detail map itself may be easily automatically generated from the input image by locating edges in the image ie. by passing the image through a derivative filter.

A tile is conceptually rigid. Its surface texture does not interact with the surface texture of the background or of any tiles it might overlap (though it would typically not overlap any other tiles). A brush stroke, on the other hand, is conceptually plastic. Its thickness and surface texture are affected by the surface texture of the background and of any other brush strokes it might overlap. Specifically, the brush stroke will tend to fill in depressions in the background, but tend itself to become thinner where it lands on peaks in the background. (The background is taken to include the effect of any brush strokes which precede the stroke in question).

In an alternative embodiment, when a brush stroke is laid down, its texture is combined with the texture of the background in one of several ways. Firstly, the stroke height is added to a proportion (say 25%) of the background height to yield the new background height, but the height is constrained not to diminish. This texturing effect will first be described with reference to Fig. 280(a) to Fig. 280(c). In Fig. 280 (a), there is illustrated 4010 an example brush cross-sectional profile of a single tile. In Fig. 280 (b), there is illustrated by way of example, a corresponding cross section of a portion of an image 4011 over which the brush pattern 4010 of Fig. 280 (a) is to be composited over. In Fig. 280 (c), there is illustrated the results of the compositing process utilizing the aforementioned rule. The portions of the curve 4013, 4014, 4016 and 4017 are taken directly from the image 4011 as the image value 4011 exceeds the corresponding brush value 4010. However, the portion 4015 is computed by adding the corresponding stroke height of the stroke 4010 to 25% of the corresponding portion of the image 4011. The background under the stroke is computed. At a particular pixel, if the background height is less than the average, then the stroke height is simply added to the background height. Thus this simulates the stroke "filling in" background depressions. If the background height is greater than (or equal to) the average, then the stroke height is added to the average. Thus the stroke is thinned by background peaks. To prevent the background from actually breaking through the stroke (ie. thinning it to zero), the height is constrained to increase by a minimum amount - the minimum stroke thickness (which may of course be zero).

Further the opacity of the stroke may be scaled (on a pixel-by-pixel basis) to account for the net thinning of the stroke which results from the texture combining algorithm. Thus where the stroke is thinned by peaks in the background is also becomes more transparent.

Once the tile/stroke position is determined, it may be manipulated further in a number of ways. For example, the position may be randomly jittered within a specified range at a specified rate to simulate natural variations in manual tile or stroke positioning. When simulating tiles rather than brush strokes, the tile jitter range is typically limited to prevent tiles from overlapping, - ie. only the width of the "inter-tile grouted gap" should be varied. When simulating brush strokes, the stroke jitter range is typically less limited since brush strokes typically do overlap.

Further, alternatively the position of each tile may be displaced according to a displacement map to simulate more regular variation in the tile or stroke position. The displacement map can consists of two channels. The first contains column displacements. The second contains row displacements.

The tile/stroke color itself may also be manipulated further for artistic effect. For example, (1) the color may be mapped to an arbitrary palette to simulate a limited set of tiles or a limited paint palette. If the color is mapped to a palette, then the overall image reproduction may be improved by error diffusing the color error locally into the remaining part of the input image. (2) The color may be randomly jittered within a specified range at a specified rate to simulate tile color imperfections or paint mixing variations. (3) The chrominance component of the color may be randomly inverted at a specified rate to simulate the style of Impressionism which utilizes color opposites.

Once the tile/stroke average global opacity is determined, it may be manipulated further for artistic effect. For example, the averaged opacity may be thresholded to give a zero opacity or 100% opacity. This then simply makes the tile absent or present. If the averaged opacity is thresholded, then the global opacity reproduction may be improved by error diffusing the opacity error locally into the remaining part of the input image. This has been found to produce important effects in an image.

A further refined embodiment is preferable implemented through suitable programming of the Artcam device. A further refined embodiment has been constructed by noting that a human painter when "rendering" an image on canvas normally works with a limited palette of paint colors, with a constrained set of brushes which produce characteristic strokes, and paints on a canvas of a particular color and texture. The painter paints strokes of different color on the canvas to that the painting is more or less true to the scene being painted, or the painter's vision of the scene. Some painting styles try to reproduce the features and colors of the scene in as much detail as possible, and are characterised by detailed brushwork, extensive paint mixing and layering. Other styles try to capture more of an impression of the feature and colors of the scene, and are characterised by simpler and more uniform brush strokes, and little paint mixing or layering. These impressionistic styles achieve brighter and more vivid color than the realistic styles, but at the expense of detail.

Computer system which automatically converts a photographic image into a "painted" rendition of that image typically places brush strokes in the output image whose color is derived in some manner from corresponding groups of pixels in the input image. The challenge in such a system lies in capturing aspects of the human painting process described above. A partial solution to producing an impressionistic rendition lies in constructing a color palette based on the colors of real artists' paints, selecting each brush stroke color from the palette to match the input image as well as possible, and error diffusing the color to maintain overall color consistency with the input image.

In a further refined embodiment, there is constructed an algorithm which error diffuses an image on the basis of the final color of the brush stroke, rather than on the basis of the raw color alone. This final color is determined by the paint color, the brush texture, and the canvas color and texture.

The input to the algorithm is the following:

* A restricted palette of paint colors.
* A set of brush strokes, each defined by an opacity map and a bump map. The use of bump maps is well known to those skilled in the art of 3 dimensional graphics rendering and is described, for instance, in Graphic Gems, Volume 4 at pages 433 - 437.
* A canvas, defined by a color image and a bump map.
* A color image to be reproduced as a painting.

The output from the algorithm is a rendition of the input image, "painted" onto the canvas with brush strokes colored from the paint palette. Because the canvas and brush strokes are textured, and these textures are combined during the painting process, the final painting may be directionally lit.

In order to achieve the painted effect, various modifications are carried out the basic tiling algorithm to achieve greater color consistency between the output image and the input image. The initial process is a brush stroke palette feedback algorithm. This process proceeds by the creation of a table of a stroke color palettes. The table 3925 as illustrated in Fig. 281 is indexed by brush stroke B1, B2, B3 etc. For each brush stroke $B_n$ in the brush stroke set and for each paint color C1 - CN in the paint palette set a stroke color S1 - S4, is determined as follows as described with reference to the flow chart of Fig. 282:

1. The paint color C1 is utilized to color the corresponding brush stroke B1 at step 3921.
2. The colored brush stroke is then composited with an empty canvas utilizing the opacity map and bump map.
3. The average color across the painted stroke is then calculated at step 3923. This average color then becomes the stroke color S1 of Fig. 281. A pointer (not shown) is then set to point from the stroke color palette entry S1 to the corresponding paint color palette entry C1.
4. The stroke color table is then sorted by color.

When painting an image, as illustrated in Fig. 283, each brush stroke is selected in the usual way. The desired color of each brush stroke is then determined from the input image in the usual way. However, the nearest paint color $C_n$ in the paint color palette is not used. Instead, the nearest stroke color $S_n$ in the stroke color palette for that brush stroke is used. Next, the corresponding paint color $C_n$ is used as the paint color for the brush stroke.

Since the range of colors available in the paint palette is limited, there is usually some difference between the desired paint color and the selected paint color $C_n$. This difference, or error can be error diffused into adjacent unvisited parts of the input image in the usual way when doing error diffused. The error as the difference between the desired color and the paint color $C_n$ is not used. Instead, the error being the difference between the desired color and the stroke color $S_n$ is used which is the average color across the painted stroke for this purpose. Note that this average color $S_n$ may differ from the average color in the stroke palette, since the real brush stroke may overlap other strokes, whereas the stroke in the stroke palette is painted on an empty canvas.

This method can be extended such that the effect of any number of painting parameters can be taken into account in the same way. This includes variations in the painting medium (oil pain, chalk, charcoal, etc.), variations in the canvas (textured paper, glass, etc.), etc. Each parameter will add a dimension to the multi-dimensional table of stroke color palettes, but the essential algorithm is unchanged.

Artists sometimes apply a mixture of two or more paints in a single brush stroke. In the impressionistic style these paints are often not well mixed, and the unmixed colors are visible as separate components of the stroke.

In the context of the aforementioned algorithm, a multi-colored brush stroke can be a generalisation of a single-colored brush stroke. It can be defined by multiple opacity maps, one for each color. These opacity maps define non-zero opacities for mutually exclusive regions of the brush stroke. Taken together they define the shape of the overall bush stroke. By introducing the following modifications to the algorithm, support multi-colored brush strokes can be provided.

When building the stroke color palette for each brush stroke, color the brush stroke with each possible combination of paint colors from the paint color palette. This is clearly only practical for small paint color palettes. Construct each stroke color palette entry in the usual way, but point each stroke color palette entry to the corresponding multiple paint color palette entries used to construct it. When painting the image, find the nearest stroke color in the stroke color palette in the usual way, but color the stroke with the corresponding multiple paint colors.

Art30 Description

Turning now to Fig. 284, there is illustrated the steps of a further refinement to the programming of an artcam device. It is assumed that the brush stroke is defined as a piecewise Bezier curve as is standard practice. The brush stroke is further assumed to have a predetermined thickness. The stroke is made up of a series of Bezier curves. The first step 4211 is to process each piecewise bezier curve. Each bezier curve is first converted to a corresponding piecewise linear curve 4212 utilizing standard techniques. Next, the "fastest" edge, whose meaning will become more apparent hereinafter, is utilized to "step along" the fastest edge in predetermine increments. The positions are utilized to composite brush strokes 4214.

Turning now to Fig. 285, there is illustrated a first Bezier curve 4220 which forms the basis of brush stroke 4221. The process of converting the Bezier curve 4220 to a corresponding series of line segments eg. 4222. The process of linearisation into piecewise linear segments is standard and well known, being covered in the usual texts. As part of the linearisation process, the normals eg. 4224 at each of the line end points are also determined. The normals can be determined from the lines adjoining a point eg. 4225 by means of interpolation between the line gradients if necessary.

Turning now to Fig. 286, there is shown an enlarged view of two line segments eg. 4230, 4231. For each point eg. 4232 the normals eg. 4233, 4234 are projected out on each side to a distance of the brush stroking width. The point 4236 having projected normals 4237, 4238. Next, the distance 4239 is measured between normals 4233, 4237 and the distance 4240 is measured between the normal 4234, 4238. A "fastest" edge is defined to be the side having the greatest distance in the sense that a body travelling along the edge 4239 would have to travel substantially faster than along the edge 4240. Hence, the fastest edge will be the convex edge of a curve.

The fastest edge is utilized to define a predetermined number of points which are parametrically spaced apart by equal amounts. The example of Fig. 286 three points eg. 4243 - 4245 which defined equally spaced intervals "in distance" along the line 4239. The number of intervals being substantially greater than that illustrated in Fig. 286.

Next, a corresponding parametric position of each of the points is determined and is utilized to determine corresponding parametric points along the line 4230. The corresponding locations being positions where brush strokes are to be placed.

Turning now to Fig. 287, there is illustrated a series of brush stamps 4246, 4247, 4248. The two stamps 4246, 4248 being utilized for the end portions of a line and the portion 4247 being continually utilized along the middle portion of a line.

The brush stamp is defined by three separate channels being a matte channel, a bump map channel and a footprint channel. The in the particular example of Fig. 287, the matte channel of the brush is illustrated. In the first technique, known as the "max" technique the matte and bump map channel are utilized to build a brush stroke

in a separate brushing buffer. In this compositing technique, a brush is built up by taking the maximum opacity ie, the opacity is replaced with a new opacity when the new opacity exceeds the old opacity presently in the buffer. The results produced are similar to the Photoshop air brushing or painting.

In a second compositing technique, hereinafter known as the "footprint" technique. The footprint channel is utilized such that the matte value is changed only if a previous brush stamp for a current line has not written to the brush buffer for that pixel. In the third technique, again the matte and the bump map channel are utilized but this time the minimum of the matte channel and the background, provided the minimum is below an absolute minimum is utilized.

The above techniques were found to give good results, especially in the simulation of water color effects. The following code segment illustrates the line following process of a further refined embodiment:

```
/*
 * $RCSfile: G2PolylineNav2.h$
 *
 * G2_PolylineNavigator2 template class definition.
 *
 * Author: Paul Lapstun, 1996/11/7
 * Copyright Silverbrook Research 1996.
 *
 * $Log$
 */
#ifndef _G2PolylineNav2_h_
#define _G2PolylineNav2_h_

#include <bool.h>
#include <G2Polyline.h>
#include <G2VectorList.h>

template<class T>
class G2_PolylineNavigator2
{
public:
                                        G2_PolylineNavigator2(G2_Polyline<T> const&
pl, G2_VectorList<T> const& tl, G2_Real halfWidth)
                                            : m_polyline(pl), m_tanList(tl),
m_halfWidth(halfWidth) { Start(); }
        bool                    AtEnd() const { return m_pi == m_polyline.end(); }
        bool                    Empty() const { return m_polyline.empty(); }
        void                    Start();
```

```
        void                        Advance(T s);
        G2_Point<T>                 Point() const;
        G2_Vector<T>                Tangent() const;

private:
        void                        SetEffectiveSegmentLength();

private:
        G2_Polyline<T> const&   m_polyline;     // polyline being navigated
        G2_VectorList<T> const&     m_tanList;      // tangent list being navigated
        G2_Real                     m_halfWidth;// half 'brush' width
        G2_Polyline<T>::const_iterator
                                        m_pi;           // polyline iterator
        G2_VectorList<T>::const_iterator
                                        m_ti;           // tangent list iterator
        G2_Point<T>                 m_p0;           // point at start of segment
        G2_Point<T>                 m_p1;           // point at end of segment
        G2_Vector<T>            m_v0;           // tangent at start of segment
        G2_Vector<T>            m_v1;           // tangent at end of segment
        G2_Vector<T>            m_n0;           // normal at start of segment
        G2_Vector<T>            m_n1;           // normal at end of segment
        T                               m_length;       // length of current segment
        T                               m_s;            // displacement along current
segment
        G2_Real                     m_t;            // parameter along current segment
};

template<class T>
inline
void
G2_PolylineNavigator2<T>::Start()
{
        // set iterators to beginning
        m_pi = m_polyline.begin();
        m_ti = m_tanList.begin();
        if (AtEnd())
                return; // empty...

        // get segment start point and tangent
```

```
		m_p0 = *m_pi++;
		m_v0 = *m_ti++;
		m_n0 = m_v0.Normal().Normalise() * m_halfWidth;
		if (AtEnd())
			return; // shouldn't happen...

		// get segment end point and tangent
		m_p1 = *m_pi;
		m_v1 = *m_ti;
		m_n1 = m_v1.Normal().Normalise() * m_halfWidth;

		// set effective segment length
		SetEffectiveSegmentLength();

		// reset displacement and parameter at start
		m_s = 0;
		m_t = 0;
}

template<class T>
inline
void
G2_PolylineNavigator2<T>::Advance(T s)
{
		if (AtEnd())
			return;

		// skip segments until s falls in current segment
		while ((m_s + s) > m_length)
		{
			// get segment start point and tangent
			m_p0 = m_p1;
			m_v0 = m_v1;
			m_n0 = m_n1;

			// get segment end point and tangent
			m_pi++;
			m_ti++;
			if (AtEnd())
```

```
                return; // genuine end...
            m_p1 = *m_pi;
            m_v1 = *m_ti;
            m_n1 = m_v1.Normal().Normalise() * m_halfWidth;

            // allow for rest of previous segment
            s -= (m_length - m_s);

            // set effective segment length
            SetEffectiveSegmentLength();

            // reset displacement at start of segment
            m_s = 0;
        }

        // move along current segment by remaining s
        // and compute parameter at new displacement
        m_s += s;
        m_t = m_s / m_length;
}

template<class T>
inline
G2_Point<T>
G2_PolylineNavigator2<T>::Point() const
{
        if (AtEnd())
                return G2_Point<T>(0, 0);
        return m_p0.Interpolate(m_p1, m_t) ;
}

template<class T>
inline
G2_Vector<T>
G2_PolylineNavigator2<T>::Tangent() const
{
        if (AtEnd())
                return G2_Vector<T>(0, 0);
        G2_Vector<T> tangent = m_v0.Interpolate(m_v1, m_t);
```

```
        return tangent;
}

template<class T>
inline
void
G2_PolylineNavigator2<T>::SetEffectiveSegmentLength()
{
        // the normals at the end points of the current line
        // segment have already been computed and scaled in
        // length by half the 'brush' width; now construct
        // vectors which join the ends of the normals; these
        // vectors are parallel to the line segment which
        // approximates the curve; the vector on the convex
        // side of the curve is longer than the line segment;
        // the vector on the concave side is shorter; choose
        // the length of the longer vector for displacement
        // stepping; note that a parameter t computed at a
        // particular displacement along the chosen vector
        // can be used directly to compute the point on the
        // line segment which approximates the curve
        G2_Vector<T> vectorA = (m_p1 + m_n1) - (m_p0 + m_n0);
        G2_Vector<T> vectorB = (m_p1 - m_n1) - (m_p0 - m_n0);
        G2_Real len2A = vectorA.LengthSquared();
        G2_Real len2B = vectorB.LengthSquared();
        if (len2A > len2B)
                m_length = sqrt(len2A);
        else
                m_length = sqrt(len2B);
}

#endif /* _G2PolylineNav2_h_ */
```

Art31 Description

Turning now to Fig. 288, there is illustrated 4301, the arrangement of a further refinement which includes an Artcam device 4302, being interconnected to a text input device 4303 which can comprise a touch pad LCD with appropriate character recognition. Alternatively, the text input device can comprise a keyboard entry device eg. 4304. A suitable form of text input device 4303 can comprise an Apple Newton (Trade Mark) device suitably adapted and programmed so as to interconnect with the Artcam device 4302. Alternatively, other forms of text

input device 4303 can be utilized. Further, the Artcard device 4305 is provided for insertion in the Artcam 4302 so as to manipulate the sensed image in accordance with the schema as illustrated on the surface of the Artcard, the manipulations being more fully discussed previously.

Turning now to Fig. 289 there is illustrated the preferred form of operation of a further refinement. In this form of operation, the Artcard 4305 is encoded with a Vark script which includes a font as defined for a roman character set and a description of how to create extra characters in this font. The description can comprise, for example, how to manipulate an outlined path so as to create new characters within the font.

The input device 4303, 4304 includes input device fonts stored therein. The input device fonts can be utilized for the display of information by the text input devices 4303, 4304, particularly in non roman character sets. Hence, the input devices 4303, 4304 can be utilized for the entry of text fields as required by the Artcard 4305. Upon entry, the outline of the font is downloaded to Artcam unit 4302 which is responsible for processing the outline in accordance with the instructions encoded on Artcard 4305 for the creation of extra characters. The characters are therefore created by Artcam device 4302 and rendered as part of the output image which is subsequently printed to form output image 4306.

Utilising this method of operation, the flexibility of the Artcam device 4302 is substantially extended without requiring the Artcam device 4302 or Artcard device 4305 to store each possible arrangement of fonts in each possible language. In this way, it is only necessary for the text input devices eg. 4303, 4304 to be country specific which substantially reduces the complexity of models which must be made available for operation of the Artcam device 4302 in a non-roman character language format.

Art33 Description

In a further refinement, it is assumed that it is desired to reproduce a camera image taken with a digital image camera and printed out on a print device such as an inkjet printer which has a predetermined dpi (say 1600 dpi) such as that disclosed in the aforementioned patent application.

Turning now to Fig. 290, such a digital imaging camera device 4501 is designed to printout images 4502 having a standard output resolution. The image 4502 being printed out by an inkjet printer device having a multi color outputs (cyan, magenta and yellow) and consists of an array of dots 4504 for each color component of the input image. When it is desired to create a copy of the outputted photograph 4502, the image is first scanned and the relevant color components are derived.

The method of a further refinement is illustrated in Fig. 291. In this method, the photographic image 4502 is scanned utilizing a linear CCD 4510 which provides for full color scanning of images into corresponding color component. Suitable linear CCD scanners are known in the art. For a description of the construction and operation of linear CCD devices, reference is made to a standard text such as in "CCD arrays, cameras and displays" by Gerald C Holst, published 1996 by SPIE Optical Engineering Press. Further, suitable sensor devices are regularly described in the IEEE Transactions on Consumer Electronics.

It is assumed that the CCD array 4510 scans an image passing under the CCD head to produce signals which are analogue to digitally converted so as to form corresponding 4504 bit digital values. The CCD array 4510 operates at 4800 dots per inch being three times the dot resolution of a photograph. The CCD 4510 can comprise three monocolor with filters, CCDs, one for each color. The rate of 4800 dpi can be achieved by utililizing a series of staggered CCD arrays, each one offset from an adjacent array.

The data values are forwarded to a frame buffer controller 4512 and stored in a frame buffer 4513 by color components. The stored image is processed to extract the position of the original dots as printed on the photograph and scanned by the CCD at the previously mentioned CCD scanning resolution of approximately three times that with which it was printed. Unfortunately, a number of defects may exist in the scanned image. These include defects associated with this scanning process including the effects of scratches and warping of the photograph 4502 in addition to a possible slight rotation of the photograph 4502 when fed through the CCD scanner 4510.

The procedure for determining the patterns and dots firstly relies on utilizing a process to extract the likely rotation of the photo 4502. In order to determine such a likely rotation, many methods can be utilized.

For example, the scanned pattern of ink dots will have certain fundamental characteristic frequencies which will be dependent upon the rotation. As the ink is printed on the photo utilizing a regular array of dots, the Fourier transform of the image can be analyzed to determine a likely rotation. Alternatively, the edges of the card can be determined from the abrupt boundary of the photo and the underneath scanning surface of the CCD. Further, an expected dot pitch of the pixels (1600 dpi) can be determined.

Starting from one border of the photograph, the scanned image of the photograph is then processed so as to determine whether a dot is located at each possible output dot location. Importantly, a method is utilized to maintain local synchronisation across the card as it is processed to determine likely output dots. It will be evident that local variations in spacings from one column to the next will be extremely minor, with the main variations occurring gradually over the length of the image.

Having determined the rotation and the likely spacing between adjacent dots, one form of processing is to keep a column of likely dot centres (herein after known as "centroid") and to update the centroids in a column by column manner. Turning to Fig. 292, there is illustrated one form of centroid processing wherein an area of an image 4520 is shown having example ink dots 4521 on the surface portion of a card. The size of each ink dot for each pixel is approximately three times the corresponding pixel sample rate. A series of centroid markers eg 4522 are kept for each column. For the centroid markers of column $C_n$+1, the centroid markers of previously calculated column and the centroid markers for the adjacent column $C_n$ are utilized to determine an initial likely centroid location. For example, in order to determine an initial position of centroid marker 4523 the centroid markers 4522, 4524 and 4525 are utilized to determine a likely location of centroid marker 4523. Once a initial likely location has been determined, the pixel values around the centroid marker 4523 are examined so as to determine whether any minute adjustment of the centroid marker 4523 is required.

The decision to move a centroid 4523 from its expected location is derived by examining pixel values around the point 4523. The examination can occur independently in the X and Y direction and the movement can also occur independently in these directions.

Many different methods could be utilized. One method for determining whether minute adjustment of the centroid 4523 is required is will now be discussed with reference to Fig. 293. In this method, it is noted that only a limited number of adjacent ink dot arrangements are possible. These possible arrangements are as illustrated 4530 in Fig. 293 which illustrates a current pixel 4531 and its two adjacent pixel 4532 and 4533. Each of the ink dots 4531 - 4533 will have approximately three corresponding pixel values as sensed by the CCD scanner. For each pixel pattern 4530 in Fig. 293, there is also shown example CCD output values 4535 that are likely CCD output values after having been digitally converted. Due to CCD sampling effects and other mechanical and photopic effects, the CCD values

4535 of corresponding dot patterns often comprise the equivalent of analogue to digitally converted Gaussian curves or portions of Gaussian curves. Hence, no abrupt edges are normally provided. However, the cross section expected can be examined against those obtained to determine the closest cross section and, provided errors are not too large, the centroid can be adjusted within limits to produce a better fit. In this way, a new centroid position can be obtained in a first dimension. Of course, the centroid adjustment problem is symmetrical in both dimensional directions and the same processing steps can be applied in the other dimension. Numerous other techniques may be utilized including more advanced techniques such as neural network training from sampled images.

Upon determining an adjusted position, the centroid 4523 (Fig. 292) is adjusted slightly and the process of centroid determination continues. From the new centroid location, the surrounding sample values are examined to determine whether a on or off ink dot value should be recorded for the particular centroid location.

Returning to Fig. 290, the recorded ink dot values are stored in an ink dot array 4516 and subsequently utilized for printing out at the same (1600 dpi) or a different resolution on a printer eg 4517 so as to produce a copy of the photograph 4518 which is ink dot for ink dot equivalent to the original photo 4511. In this way, a high quality photographic copies is provided for making copies from the original photographs without having to utilize negatives or to digitally store the image separately.

Of course, many refinements may be possible, including utilization of customised real time pipelined architectures for speeding up processing and eliminating the need to store large data sets in the frame buffers or the like.

Art34 Description

In a further refinement, the Artcam device is modified so as to include a two dimensional motion sensor. The motion sensor can comprise a small micro-electro mechanical system (MEMS) device or other suitable device leave to detect motion in two axes. The motion sensor can be mounted on the camera device and its output monitored by the Artcam central processor.

Turning now to Fig. 294, there is illustrated a schematic of the preferred arrangement of a further refinement. The accelerometer 4601 outputs to the Artcam central processor 4602 which also receives the blurred sensed image from the CCD device. The Artcam central processor 4602 utilizes the accelerometer readings so as to determine a likely angular velocity of the camera when the picture was taken. This velocity factor is then utilized by a suitably programmed Artcard processor 4602 to apply a deblurring function to the blurred sensed image 4603 thereby outputting a deblurred output image 4604. The programming of the Artcard processor 4602 so as to perform the deblurring can utilize standard algorithms known to those skilled in the art of computer programming and digital image restoration. For example, reference is made to the "Selected Papers on Digital Image Restoration", M. Ibrahim Sezan, Editor, SPIE Milestone series, volume 74, and in particular the reprinted paper at pages 167-175 thereof. Further, simplified techniques are shown in the "Image Processing Handbook", second edition, by John C. Russ, published by CRC Press at pages 336-341 thereof.

Art38 Description

Turning initially to Fig. 295, there is illustrated 5010 a form of an auxiliary printer as construction in accordance course with a further refinement. The printer 5010 is designed to take a print roll 5011 for the insertion into a cavity 5012. The print roll 5011 is inserted in the cavity 5012 and contains rollers eg. 5013, 5014 which pinch an internal paper roll an feed it, on demand, past a print head 5015 for the printing of images. Ideally,

the print roll 5013 includes print media contained therein in addition to a consumable ink supply with the ink supply being consumed by the print head 5015 in the printing out of images.

The printer 5010 can be constructed from a suitable reworking of the artcam embodiment wherein the printer 5010 communicates with the computer 5024 via a USB port. The printer unit 5018 can be a plastic moulded case and includes an internal print head 5015 for the printing out of images.

The print head 5015 is interconnected to a print head chip 5020 by means of automated bonding techniques. The print chip 5020 can be mounted on a flexible circuit board so as to reduce the size of the printer unit 5018. The printer head chip can be constructed in accordance with the latest semiconductor fabrication techniques and include associated memory chips etc. for the production of images from a description forwarded to the printer unit 5018 via a USB communications cable 5022. Of course, many other forms of interfacing could be utilized. The printer head chip can be as previously described.

Hence, when it is designed to print out an image, the computer system 5024 interconnecting to the printer unit 5018 forwards the page description to the print unit 5018 via cable 5022. The print chip 5020 prepares the image for printing which is subsequently printed out on printer head 5015. The print roll 5011 thereby supplies a convenient form of consumable print media including ink and upon utilization by the printer unit 5018 the print roll is disposed of a new print roll inserted into cavity 5012.

Art40 Description

In a further adaptation, the camera system and printing system is dispensed with and replaced with a large screen reader. Artcards can then be provided having on one surface there printed an indicator of the information on a back surface. For example, the artcard could include a book's contents or a newspaper content. An example of such a system is as illustrated in Fig. 296 wherein the artcard 5210 includes a book title on one surface with the second surface having the encoded data printed thereon. The card 5210 is inserted in the reader 5212 which includes a flexible display 5213 which allows for the folding up of card reader 5212. The card reader 5212 includes display controls 5214 which allow for paging forward and back and other controls of the card reader 5212.

It can therefore be seen that the arrangement of Fig. 296 provides for an efficient distribution of information in the forms of books, newspapers, magazines, technical manuals, etc.

It would be appreciated by a person skilled in the art that numerous variations and/or modifications may be made to the present invention as shown in the specific embodiment without departing from the spirit or scope of the invention as broadly described. The present embodiment is, therefore, to be considered in all respects to be illustrative and not restrictive.

Art47 Description

A further refinement will be discussed with reference to the process of "painting" an image onto a canvas utilizing simulated brush strokes. It is assumed that bump maps techniques are utilized and, in particular, each image utilized has an associated bump map defining a texture of the surface of the image.

In Fig. 297, there is illustrated an example round brush stroke bump map 5901; with Fig. 298 illustrating a corresponding section through the line A - A' of Fig. 297 and includes a general profile 5903 of the surface of the generally round brush stroke 5901. It is assumed that it is desired to render a brush stroke having a bump map as illustrated in Fig. 297 onto a generally hessian shaped "canvas" image whose bump map is as illustrated in Fig.

299 with again, the height through the line B - B' being illustrated in Fig. 300. The height consisting of a series of undulating peaks eg. 5906.

In a further refinement, the bump maps of Fig. 298 and Fig. 300 are combined to produce a final bump map in a variable manner in accordance with a supplied stiffness factor. The stiffness factor being designed to approximate the effects of using a"stiff" or "flexible" paint. In Fig. 301, there is illustrated the example of a combination of the two bumps of Fig. 298 and Fig. 300 wherein they are combined for a brush paint having a high stiffness. In Fig. 302, there is illustrated example of a combination of the two bump maps of Fig. 298 and Fig. 300 when a low stiffness paint has been utilized.

In a further refinement, when combining bump maps, the bump maps are not simply added, since the brush stroke which consists of paint will have a certain plasticity and will therefore fill in depressions in the background. Hence, in order to achieve the effects of Fig. 301 and Fig. 302, the brush stroke bump map is added to a low-pass filtered version of the background bump map. The low-pass filter has its radius determined by a paint stiffness factor. The higher the stiffness factor, the wider the filter radius and hence the less the background surface texture showing through. The low-pass filter radius can also be scalable by the relative brush stroke thickness so the filtered background surface texture will converge with the actual background surface texture where the brush stroke thins to nothing ie. mostly along the edges.

Turning now to Fig. 303 and Fig. 304, there is shown an example of this process with Fig. 303 illustrating the initial background bump map. In Fig. 304, there is shown the combined background bump maps with Fig. 304 showing a high stiffness paint 5912 on top of a low-pass filtered version of the bump map 5911 with Fig. 305 showing a low stiffness paint 5914 on top of the low-pass filtered bump map 5913.

Art48 Description

In a further refinement, the color gamut of an input image is "morphed" to the color gamut of an output image wherein the output color gamut can be arbitrarily determined by means of experimentation with different output color gamuts. A further refinement is ideally utilized as a pre-processing step to further artistic manipulation such as replacing the image with brush strokes of particular style wherein the brush stroke derive their color from the image and the brush strokes having colors related to the target gamut.

The steps in producing a final output image are as illustrated 6010 in Fig. 306. The first step 6011 is the input of an arbitrary input image, preferably as sensed by the Artcam device. The colored gamut of the input image is then"Morphed" or "Warped" 6012 so as to produce an output image having a predetermined color gamut range. The morphing process being further described hereinafter. Once the output color gamut is produced, the next step is to apply a suitable brush stroking technique. The brush stroking technique being a post processing step able to be subjected to substantial variation, the actual form of brush stroking utilized not being essential to the present invention with techniques being disclosed previously. The brush stroking technique is applied 6013 so as to produce an output image 6014 having a restricted colored gamut interpretation of the input image.

Turning now to Fig. 307, there is provided an example of the gamut mapping or morphing problem. It is assumed that a single color space is utilized defining the universe of possible color values with the universal color space 6021 being utilized which is a L*a*b* color space. Within this color space, the input sensor is able to sense a certain range of color values 6022. It is further assumed that the output printing device is able to output colors of a particular range of printer gamut 6023. The printer gamut may be small or larger than the input sensor color

gamut 6022. An artistic gamut 6024 is defined in accordance with a particular style. The artistic gamut 6024 can be meticulously constructed through the utilization of various techniques. For example, a desired output image can be scanned and a histogram of colors built up so as to include a certain range within the L*a*b* color space. It would be understood by those skilled in the art of computer graphics that Fig. 307 can be interpreted as slices trough a 3 dimensional L*a*b* color space at predetermined intensity values or by means of 3 dimensional volumes within the color space.

It is therefore necessary to map the colors within input sensor color space 6022 to the artistic gamut 6024. Further, the artistic gamut 6024 may unfortunately contain "out of gamut" colors for a particular output printer gamut 6023. In such cases, it will be further necessary to map the artistic gamut 6024 so that the output colors fall within the printer gamut 6023.

Turning now to Fig. 308, the primary requirement is for the Artcard to include a 3 dimensional look up table eg. 6030 which maps L*a*b* values 6031 to L*a*b* output values 6032. Preferably, the 3 dimensional lookup table 6030 is provided in a compact form with only certain points being defined for 3 dimensional mapping and trilinear interpolation being utilized for the mapping of intermediate values between defined points.

Turning to Fig. 309, there is illustrated a more general example of the process of applying a gamut morphing from one substantially arbitrary input gamut space 6040 to a second desired output gamut target space 6041. Potentially, the gamut mapping process must deal properly with the points eg. 6042 which lie within the input color gamut space 6040 but outside the output color gamut space 6041. The 3D color lookup table 6030 has a size of $(2^n+1)^3$, where n can range from 1 to the maximum color component precision (i.e. typically 8 bits). An arbitrary warp function can be encoded in the lookup table, which gives a high degree of flexibility to the overall algorithm. The use of the table also results in the performance of the algorithm being independent of the encoded warp function which can be separately prepared. The algorithm has predictable performance for arbitrary warp functions and it does not require the warp function to be continuous, therefore resulting in the algorithm being robust for arbitrary warp functions. Hence, the per-pixel processing is fixed and reasonably simple, and is suitable for hardware implementation.

The lookup table consists of a 3D array of integer or real-valued output color coordinates arranged in input coordinate order. It thus encodes a forward warp function.

The image color warp required is computed in output image order, one pixel at a time. The algorithm requires random access to the 3D color lookup table, but since input colors vary smoothly in space, the random access is typically coherent. It requires sequential access to the input and output images.

The essential tri-linear warp algorithm is embodied in the following pseudo-C++ code:

```
for (int row = 0; row < height; row++)
{
    for (int col = 0; row < width; col++)
    {
        outputImage[row][col] = Lookup(table, inputImage[row][col]);
    }
}
```

```
Colour
Lookup(LookupTable& table, Colour& color)
{
    // compute indices
    int fracPrecision = colorPrecision - log2(table.nSamples);
    int i = color[0] >> fracPrecision;
    int j = color[1] >> fracPrecision;
    int k = color[2] >> fracPrecision;

    // compute interpolation factors
    int one = (1 << fracPrecision);
    int mask = one - 1;
    double f = (double)(color[0] & mask) / one;
    double g = (double)(color[1] & mask) / one;
    double h = (double)(color[2] & mask) / one;

    // trilinearly interpolate
    Colour c000 = table[i+0][j+0][k+0];
    Colour c001 = table[i+0][j+0][k+1];
    Colour c00 = Interpolate(c000, c001, h);
    Colour c010 = table[i+0][j+1][k+0];
    Colour c011 = table[i+0][j+1][k+1];
    Colour c01 = Interpolate(c010, c011, h);
    Colour c0 = Interpolate(c00, c01, g);

    Colour c100 = table[i+1][j+0][k+0];
    Colour c101 = table[i+1][j+0][k+1];
    Colour c10 = Interpolate(c100, c101, h);
    Colour c110 = table[i+1][j+1][k+0];
    Colour c111 = table[i+1][j+1][k+1];
    Colour c11 = Interpolate(c110, c111, h);
    Colour c1 = Interpolate(c10, c11, g);

    return Interpolate(c0, c1, f);
}
```

The gamut compression process seeks to map the colors in a source gamut to colors in a smaller target gamut in such a way that color differences in the source gamut are retained and perceptible color shifts are

minimised. Efficient gamut compression can be carried out using the lookup-table-driven process previously described, since the computation of the gamut compression is decoupled from the color warping.

The gamut compression algorithm involves the construction of a 3D lookup table and is embodied in the following pseudo-code described with reference to Fig. 309:

```
int fracPrecision = colorPrecision - log2(nSamples);
for (int i = 0; i < nSamples; i++)
{
    for (int j = 0; j < nSamples; j++)
    {
        for (int k = 0; k < nSamples; k++)
        {
            1. for constructing the lookup table at the color point at (i << fracPrecision,j << fracPrecision, k
            << fracPrecision);
            2. find nearest point 6043 on the luminance axis of the Lab space within the target gamut 6041
            of Fig. 309;
            3. construct a vector from the color point 6042 to a point on luminance axis 6043;
            4. compute the distance dp of the color point 6042 from the luminance axis 6043;(i.e. the length
            of the vector);
            5. compute the intersection 6045 of the vector with source gamut polyhedron 6040;
            6. compute the intersection 6046 of the vector with the target gamut polyhedron 6041;
            7. compute the distance dt from the luminance axis 6043 to the target gamut boundary 6046
            along the vector;
            8. compute the distance from the luminance axis 6043 to the source gamut boundary 6045 along
            the vector;
            9. compute the compression factor as ratio of target distance dt 6043 - 6046 to the source
            distance ds 6043 - 6045;
            10. compute the scale factor as the ratio of the color point distance dp 6043 - 6042 to source
            distance ds 6043 - 6045;
            11. scale the compression factor by the scale factor (so that in-gamut colors compress
            progressively less);
            12. scale the color point distance dp by the compression factor
            13. compute the point on the vector which is the compressed distance from the luminance axis;
            14. write this point to the lookup table at (i, j, k);
        }
    }
}
```

If either the source or target gamut is known only in palette form, then the gamut polyhedron can be computed from the convex hull of the palette points.

Turning now to Fig. 310, the above process can be adapted to map any source gamut 6050 to a target gamut 6051 utilizing a process heretoafter called gamut morphing which has particular application in the creation of artistic effects. Gamut morphing is used to directly control the mapping of colors in the source gamut to colors in the target gamut.

Just like gamut compression, gamut morphing may be used to artificially constrain the gamut of an image to simulate a particular artistic style. Just like gamut compression, efficient gamut morphing can be carried out using the lookup-table-driven algorithm, since the computation of the gamut morph is decoupled from the color warping process.

With gamut morphing, a number of source gamut colors e.g. 6053 are mapped directly to the same number of corresponding target gamut colors e.g. 6054. The remaining target colors intermediate of the mapped targets in the lookup table are computed as a weighted sum of the specified target colors (e.g. 6054). The target colors are preferably weighted by the inverse squared distance of the corresponding source colors from each lookup table point.

The gamut morphing algorithm is embodied in the following pseudo-code:

```
int fracPrecision = colorPrecision - log2(nSamples);
for (int i = 0; i < nSamples; i++)
{
    for (int j = 0; j < nSamples; j++)
    {
        for (int k = 0; k < nSamples; k++)
        {
            ... construct color point at (i << fracPrecision,
                    j << fracPrecision, k << fracPrecision)
            ... initialise the weighted sum to zero
            ... initialise the sum of weights to zero
            for (int m = 0; m < sourceGamut.size(); m++)
            {
              compute the distance from point to a current source gamut point corresponding to m;
              compute the inverse of the distance squared;
              call the calculation from step 2 the weight, and add it to the sum of weights
              scale the corresponding target gamut point by the weight and add it to the weighted sum
            }
            ... divide the weighted sum by the sum of weights
            ... write this point to the lookup table at (i, j, k)
        }
    }
}
```

It will be evident to those skilled in the art of computer graphics that the aforementioned technique can be utilized to initially restrict the gamut of an image to predetermined areas. Subsequently, brush stroking filters can be applied to the restricted gamut image to produce effects similar to those provided, for example, by the "Pointillisme" techniques.

Art53 Description

The basics of the aforementioned Artcam system are indicated 6501 in Fig. 311. The Artcam system 6501 relies on an Artcam 6502 which takes Artcards 6503 as an input. The Artcard 6503 includes encoded information for manipulation of an image scene 6504 so as to produce an output photo 6505 which contains substantial manipulation in accordance with the encoded instructions, of Artcard 6503. The Artcards 6503 are designed to be extremely inexpensive and contain on one surface the encoding information and on the other surface a depiction of the likely effect which will be produced by the Artcard 6503 where inserted in Artcam 6502.

In accordance with the method of a further refinement, as shown in Fig. 312, a large number of Artcards 6510 are prepared and distributed in packs. Each pack relates to clothing wear of a specific size and includes images eg. 6511 of models having clothing apparel 6512 on to which an image captured by the camera will be mapped. The mapping can be for different items of apparel on different Artcards 6510. One form of mapping algorithm is as illustrated 6520 in Fig. 313 wherein the input image 6504 is first warped 6521 utilizing a warp map which maps the image to a repeating tiling pattern that produces attractive warping effects. Of course, many other forms of algorithms could be provided for producing an attractive form of material with the algorithm being provided on Artcard 6503 (Fig. 311).

Next, a second warp 6522 can be provided for warping the output of first warp map 6521 onto the specific model image in the Artcard. Therefore, warp 6522 will be Artcard specific. The result can then be output 6523 for printing as an Artcam photo 6505. Hence, a user is able to point a Artcam 6502 at a desired image 6504 and produce Artcam photo 6505 which has a manipulated version of the image based upon a models item of fashion apparel or garment. This process can be continued until a desirable result is produced.

Next, as indicated in Fig. 314, when a final selection of fabric has been made, the Artcam 6502 can be interconnected 6503 by its USB port to a fabric printer 6534 which can comprise an ink jet fabric printer and associated drive controller electronics etc. Either the Artcam 6502 or the ink jet printer 6534 can be programmed to print out on fabric 6535 the garment pieces eg. 6536 having on the surface 6537 thereof the original warped image so as to produce a corresponding garment corresponding to that depicted by the model on the Artcard.

The output fabric can include tab portions eg. 6538 for alignment and border regions eg. 6539 in addition to instructions 6540 for joining the garment pieces together. Preferably, the output program includes providing for warp matching of boarder regions so as to present a continuous appearance on the garment cross seams. Additionally, a user interface could be provided for utilizing the same Artcard with many different output sizes so as to taken into account different shaped bodies. By utilization of Artcam technology, a system can be provided for customised production of garments and rapid depiction of the likely results by means of utilization of the Artcam device 6502.